

# Apunte de Ingeniería del Software II - Segunda Parte

Tomás C.

Segundo cuatrimestre, 2022

## Acerca de este apunte

Este es un compilado con los conceptos teóricos de la materia “Ingeniería de Software II”, escrito en base a las clases teóricas (tanto clases presenciales, como videos, como apuntes propios sobre las clases) del segundo cuatrimestre de 2022.

La materia se dividió en dos partes: testing automatizado (a cargo de Juan Pablo Galeotti) y verificación de sistemas concurrentes (a cargo de Sebastián Uchitel), y de esa manera también se divide este apunte. En cuanto a la estructura, cada sección corresponde aproximadamente a una clase teórica de la materia. Esta es la **segunda parte** del apunte, correspondiente a la segunda mitad de la materia (sistemas concurrentes).

Esto no pretende ser un *resumen* (en el sentido de acortar las cosas, como se puede ver por la cantidad de páginas) de los contenidos de la materia, sino que la idea de este apunte es ser una recopilación de lo estudiado en clase, que sea útil para preparar el examen final. Dicha utilidad podría no existir más pasadas las fechas de examen del verano 2022/2023, dado que la materia suele cambiar su contenido.

En el caso de que este apunte sea utilizado por otras personas, advierto que puede contener errores conceptuales, errores de redacción, errores de *tipeo*, errores de ortografía (i.e. “herrorez”) o gramática, errores adentro de errores, etcétera.

---

Título completo: *Apunte para el examen final de Ingeniería del Software II.*

Autor: Tomás C.

Fecha de finalización: 7 de diciembre de 2022.

# Índice

<b>II Verificación de Software Concurrente</b>	<b>1</b>
<b>7. Modelado de Sistemas Concurrentes</b>	<b>1</b>
7.1. Programas secuenciales vs. concurrentes . . . . .	1
7.2. Modelos de concurrencia . . . . .	2
7.3. Labelled Transition Systems (LTS) . . . . .	2
7.3.1. Definición . . . . .	2
7.3.2. Ejecuciones y trazas . . . . .	3
7.3.3. Composición paralela . . . . .	4
7.4. Finite State Processes (FSP) . . . . .	4
7.4.1. El proceso deadlock . . . . .	4
7.4.2. Acción prefijo . . . . .	5
7.4.3. Recursión . . . . .	5
7.4.4. Subprocesos . . . . .	5
7.4.5. Alternativas . . . . .	6
7.4.6. Composición en paralelo . . . . .	6
7.4.7. Elección no determinística . . . . .	8
7.4.8. Extensión del alfabetos . . . . .	9
7.4.9. Ocultamiento de acciones . . . . .	9
<b>8. Semántica de FSP y LTS</b>	<b>10</b>
8.1. Algunas propuestas para la equivalencia de LTS . . . . .	10
8.1.1. Isomorfismo . . . . .	10
8.1.2. Isomorfismo <i>refinado</i> . . . . .	10
8.1.3. Igualdad de ejecuciones . . . . .	11
8.1.4. Igualdad de trazas . . . . .	11
8.2. Bisimulación . . . . .	11
8.3. Relaciones de bisimulación fuerte . . . . .	12
8.3.1. El juego de la bisimulación fuerte . . . . .	13
8.3.2. Análisis de bisimulación . . . . .	13
8.4. Relaciones de bisimulación débil . . . . .	14
8.5. Consecuencias de la congruencia . . . . .	16
<b>9. Análisis de Sistemas Concurrentes</b>	<b>18</b>
9.1. Validación . . . . .	18

9.2.	Verificación . . . . .	18
9.3.	Deadlock . . . . .	19
9.3.1.	Detección de deadlock . . . . .	19
9.3.2.	Soluciones para deadlock . . . . .	19
9.4.	Exclusión mutua y estados de error . . . . .	20
9.5.	Observadores . . . . .	20
9.5.1.	Observadores en FSP . . . . .	21
9.6.	Safety y Liveness . . . . .	21
9.6.1.	Composicionalidad de safety . . . . .	21
9.6.2.	Progreso en FSP . . . . .	22
9.6.3.	Prioridad de acciones . . . . .	23
9.6.4.	Composicionalidad de liveness . . . . .	25
<b>10.</b>	<b>Model Checking usando LTL</b>	<b>26</b>
10.1.	Lógica modal proposicional . . . . .	26
10.1.1.	Semántica . . . . .	26
10.2.	Lógica temporal lineal . . . . .	27
10.3.	Semántica de LTL . . . . .	27
10.4.	Vínculo entre LTL y LTS . . . . .	29
10.5.	Algoritmo de verificación para programas concurrentes . . . . .	29
10.5.1.	Autómatas de Büchi . . . . .	30
10.5.2.	Autómatas de Büchi generalizados . . . . .	32
10.5.3.	Conversiones entre autómatas de Büchi generalizados y no generalizados	32
10.5.4.	Conversión de LTL a autómata de Büchi . . . . .	33
10.5.5.	Conversión de LTS a autómata de Büchi . . . . .	59
10.5.6.	Intersección de autómatas de Büchi . . . . .	60
10.5.7.	Chequeo de vacuidad de lenguaje . . . . .	60
<b>11.</b>	<b>Model Checking usando CTL</b>	<b>61</b>
11.1.	Lógicas temporales de <i>branching</i> . . . . .	61
11.2.	Computational Tree Logic . . . . .	61
11.3.	Semántica de CTL . . . . .	64
11.4.	Expresividad de CTL y LTL . . . . .	65
11.4.1.	CTL >LTL . . . . .	65
11.4.2.	LTL >CTL . . . . .	66
11.5.	Algoritmo de verificación para programas concurrentes . . . . .	66
11.5.1.	Una base de operadores CTL . . . . .	67

11.5.2. Semántica CTL con respecto a una estructura de Kripke . . . . .	67
11.5.3. El algoritmo . . . . .	67
11.5.4. Limitaciones del model checking explícito . . . . .	69

## Parte II

# Verificación de Software Concurrente

## 7. Modelado de Sistemas Concurrentes

### 7.1. Programas secuenciales vs. concurrentes

En esta segunda parte de la materia, se cambia el foco desde los programas secuenciales a los programas **concurrentes**, **reactivos** (se ejecutan en múltiples *threads*). Así, deja de ser tan importante lo que un programa retorne, y pasa a ser más relevante su **comportamiento observable**, y su **aspecto reactivo**: lo que haga durante su ejecución.

Un programa secuencial tiene un único *thread* de control, mientras que un programa concurrente posee múltiples *threads* de control, permitiendo realizar múltiples cómputos *en paralelo*.

Ventajas de la concurrencia:

- Mejora de performance en hardware paralelo.
- Mejora de throughput.
- Mejora de tiempo de respuesta de una aplicación.
- Más apropiado para interactuar con el ambiente, y reaccionar a múltiples eventos.

Dificultades de la concurrencia:

- Construir correctamente un programa.
- Reproducir errores (que un error ocurra o no puede depender del orden en el que se ejecuten las cosas, y este no siempre es el mismo).
- Testear programas y razonar sobre ellos.

Recuerdo: diferencias entre concurrencia, paralelismo y distribución.

- **Concurrencia** es el procesamiento lógicamente en simultáneo, no necesariamente implica varias unidades de cómputo, y requiere de *ejecución intercalada* (interleaving) en caso de contar con una única unidad de procesamiento.
- **Paralelismo**: procesamiento que ocurre físicamente en simultáneo, usando múltiples unidades de cómputo.
- **Distribución**: consiste en el procesamiento paralelo entre unidades distribuidas en distintos lugares físicos, típicamente conectadas por una red.

El foco estará puesto entonces sobre los **sistemas reactivos** (y concurrentes), que interactúan con el entorno. Lo interesante será justamente tal interacción, basada en estímulos, y no tanto en la eventual postcondición de un sistema. Asimismo, las propiedades a estudiar serán normalmente **temporales**, sobre lo que ocurra durante la ejecución.

Son ejemplos de sistemas reactivos:

- Sistemas de control de procesos físicos donde el entorno son sensores y actuadores.
- Sistemas que interactúan con humanos.
- Sistemas que interactúan con otros sistemas reactivos.

Por lo tanto, las cuestiones de interés no serán sólo aquellas compartidas con los programas secuenciales (invariantes, terminación, etc.), sino también propiedades propias de los sistemas concurrentes, como deadlock, livelock, interferencia, fairness, etc.

## 7.2. Modelos de concurrencia

Lo primero que se puede buscar para modelar un sistema concurrente es un **lenguaje adecuado**. En su versión más primitiva, esto son las **álgebras de proceso**; éstas estudian las teorías matemáticas que permiten definir lenguajes sobre los que se describen las interacciones entre agentes concurrentes. Se definen propiedades de los sistemas a partir de las descripciones en las álgebras de proceso.

De aquí surge CSP (Communicating Sequential Processes), un lenguaje formal usado para describir patrones de interacción en sistemas concurrentes<sup>1</sup>. A su vez, de este lenguaje deriva el lenguaje que se usa en la materia: **FSP** (Finite State Processes), que *hereda* de CSP, pero asegurando que toda expresión resulte en un modelo con una **cantidad finita de estados**, de manera tal que se facilite su análisis.

Existe un formalismo *gráfico* que se combina con FSP para modelar sistemas concurrentes: los **LTS** (Labelled Transition System, o *sistema de transiciones etiquetadas*). Se trata de autómatas finitos (con estados y transiciones entre ellos, etiquetadas por acciones, y un estado inicial), con la particularidad de que se puede establecer una suerte de *paralelismo* entre los términos de FSP y los diagramas LTS (como se verá más adelante).

## 7.3. Labelled Transition Systems (LTS)

### 7.3.1. Definición

En un LTS:

- Las **etiquetas** representan interacciones de un proceso con su entorno.
- Existe una acción especial  $\tau$  que modela **cómputo interno** del proceso, no observable desde el entorno.
- La cantidad de estados es **finita**.

**Definición 7.1.** Sea *Estados* el universo de estados, *Act* el universo de acciones observables, y  $Act_\tau = Act \cup \{\tau\}$ . Un LTS es una 4-upla  $P = \langle S, A, \Delta, s_0 \rangle$ , donde:

- $S \subseteq \text{Estados}$  es un conjunto finito de estados.
- $A \subseteq Act_\tau$  es un conjunto de etiquetas.

---

<sup>1</sup>Las funcionalidades de concurrencia del lenguaje Go están parcialmente basadas en CSP

- $\Delta \subseteq S \times A \times S$  es un conjunto de transiciones etiquetadas entre estados.
- $s_0 \in S$  es el estado inicial.

Se define el alfabeto de comunicación de  $P$  como  $\alpha P = A \setminus \{\tau\}$ .

Intuitivamente, el alfabeto de un LTS son todas las formas en que un LTS (es decir, el proceso representado por este) puede interactuar con su entorno, mientras que las acciones especiales  $\tau$  representan el cómputo interno que pueda existir en un proceso.

### 7.3.2. Ejecuciones y trazas

Las **ejecuciones** de un LTS son las secuencias que representan cómo evoluciona el estado del LTS, y qué acciones suceden en cada cambio de estado. A partir de una ejecución se puede derivar la **traza** de un proceso, que es la secuencia de acciones de la ejecución (sin los estados).

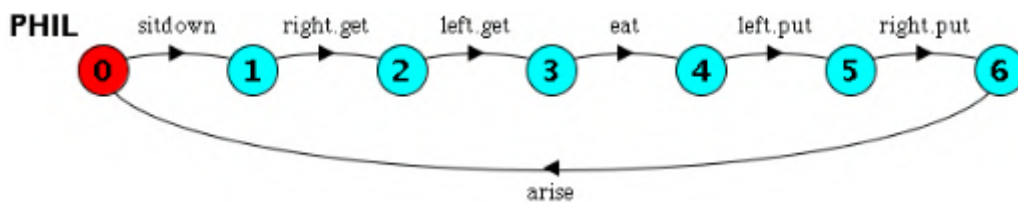
**Definición 7.2.** Una ejecución de un LTS  $P = \langle S, A, \Delta, s_0 \rangle$  es una secuencia  $s_0, \ell_0, s_1, \ell_1, \dots$ , donde para cada  $i \geq 0$  se tiene que  $\langle s_i, \ell_i, s_{i+1} \rangle \in \Delta$ .

Una secuencia  $\ell_0, \ell_1, \dots$  es una traza de  $P$  si es proyección de una ejecución de  $P$  sobre  $\alpha P$ .

**Ejemplo 7.1.** Sea el siguiente LTS que representa a un individuo en una instancia del clásico problema de los filósofos que cenar:

$PHIL = \langle \{0, 1, 2, 3, 4, 5, 6\},$   
 $\{sitdown, right.get, left.get, eat, left.put, right.put, arise\},$   
 $\{(0, sitdown, 1), (1, right.get, 2), (2, left.get, 3), (3, eat, 4), (4, left.put, 5), (5, right.put, 6), (6, arise, 0)\},$   
 $0 \rangle$

Puede ser representado por el siguiente diagrama:



Un ejemplo de ejecución es  $0, sitdown, 1, right.get, 2, left.get, 3, eat, 4, left.put, 5, right.put, 6, arise, 0, sitdown, \dots$

Un ejemplo de traza es  $sitdown, right.get, left.get, eat, left.put, right.put, arise, sitdown, \dots$

*Observación.* Dos LTS distintos pueden tener el mismo diagrama, pues puede haber acciones que ocurran en uno pero no en el otro, a pesar de ser parte del conjunto de potenciales acciones. Esto es el **alfabeto potencial** de comunicación de un LTS, que es ocultado por la representación gráfica.



### 7.3.3. Composición paralela

**Definición 7.3.** Sean los LTS  $M = \langle S_M, A_M, \Delta_M, s_0^M \rangle$  y  $N = \langle S_N, A_N, \Delta_N, s_0^N \rangle$ . La *composición paralela* ( $\parallel$ ) es un operador simétrico tal que  $M \parallel N = N \parallel M$  es el LTS  $\langle S_M \times S_N, A_M \cup A_N, \Delta, (s_0^M, s_0^N) \rangle$ , donde  $\Delta$  es la relación más chica que satisface las siguientes reglas, con  $\ell \in A_M \cup A_N$ .

$$\frac{(s, \ell, s') \in \Delta_M}{((s, t), \ell, (s', t)) \in \Delta} \ell \notin \alpha N \qquad \frac{(t, \ell, t') \in \Delta_N}{((s, t), \ell, (s, t')) \in \Delta} \ell \notin \alpha M$$

$$\frac{(s, \ell, s') \in \Delta_M, (t, \ell, t') \in \Delta_N}{((s, t), \ell, (s', t')) \in \Delta} \ell \in \alpha M \cap \alpha N$$

El *LTS compuesto* tiene como estados a los pares de estados de los LTS involucrados, tiene como acciones a todas las acciones de los mismos, y su estado inicial es el par de los estados iniciales.

Las transiciones están dadas por la relación  $\Delta$ , que puede evolucionar según cómo evolucionen las correspondientes relaciones de los componentes (independientemente), o de manera sincronizada en caso de que una acción (etiqueta) sea común a los dos componentes.

La cantidad de estados alcanzables de la composición no necesariamente es el producto de las cantidades de estados alcanzables de los estados componentes, ya que puede ocurrir que el comportamiento de un componente **restrinja** al de otro. Esto puede causar que algunos estados no sean alcanzables en el *LTS compuesto*.

## 7.4. Finite State Processes (FSP)

Se verán los principales términos disponibles en el lenguaje FSP. Hay una guía de referencia en <sup>2</sup>, y más referencias sobre semántica y sintaxis en <sup>3</sup>.

En FSP no existe una distinción entre inputs y outputs: lo que representa un evento es que hubo una interacción, abstrayendo el hecho de qué entidad la haya disparado. Tampoco existe la noción de parámetros, sino que todo es codificado como etiquetas y transiciones.

Lo que más interesa en FSP no son los estados, o lo que suceda con ellos (e.g. salida de un programa), sino las etiquetas, las acciones que pueden realizar los procesos para interactuar entre ellos y con el entorno. Los procesos emiten mensajes, sin saber *con quién hablan*, y la comunicación entre procesos se establece mediante etiquetas compartidas.

### 7.4.1. El proceso deadlock

STOP es un proceso que es incapaz de interactuar con su entorno, siendo el siguiente su LTS, que no tiene transiciones habilitadas:

<sup>2</sup><https://www.doc.ic.ac.uk/~jnm/book/ltsa/Appendix-A-2e.html>

<sup>3</sup><https://www.doc.ic.ac.uk/~jnm/LTSdocumentation/FSP-notation.html>



Figura 1:  $lts(\text{STOP}) = \langle \{s\}, \{\}, \{\}, s \rangle$

#### 7.4.2. Acción prefijo

Si  $x$  es una acción y  $P$  es un proceso, entonces  $(x \rightarrow P)$  describe un proceso que inicialmente interactúa a través de la acción  $x$ , y luego se comporta como  $P$ . Por ejemplo:

**UnicaVez = (uno  $\rightarrow$  STOP) .**

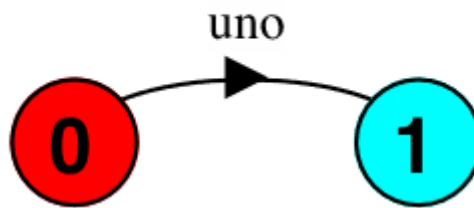


Figura 2:  $lts(\text{UnicaVez}) = \langle \{s, p\}, \{\text{uno}\}, \{(s, \text{uno}, p)\}, s \rangle$

#### 7.4.3. Recursión

La repetición de comportamiento se modela con recursión. Por ejemplo, el proceso **SWITCH** = (on  $\rightarrow$  off  $\rightarrow$  SWITCH) modela un interruptor que se puede encender y apagar tantas veces como se quiera.

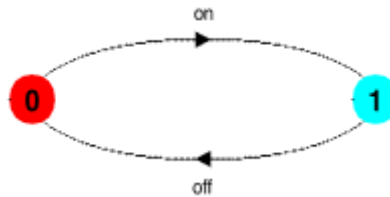


Figura 3:  $lts(\text{Switch}) = \langle \{s, p\}, \{\text{on}, \text{off}\}, \{(s, \text{on}, p), (p, \text{off}, s)\}, s \rangle$

#### 7.4.4. Subprocesos

Los subprocessos son definiciones de procesos locales a un proceso. Pueden servir para definir el comportamiento de un proceso en base a los estados de su LTS, donde cada estado representa un subprocesso. Se utiliza la coma ',' para introducir un subprocesso, y el punto '.' para terminar la definición del proceso.

Por ejemplo, los siguientes procesos son equivalentes, y su LTS es el de la Figura 3.

1. SWITCH = OFF,  
OFF = (on  $\rightarrow$  ON),  
ON = (off  $\rightarrow$  OFF) .

2. SWITCH = OFF,  
OFF = (on -> (off -> OFF)).
3. SWITCH = (on -> off -> SWITCH).

#### 7.4.5. Alternativas

Si  $x$  e  $y$  son acciones, entonces  $(x \rightarrow P \mid y \rightarrow Q)$  describe un proceso que inicialmente es capaz de interactuar a través de las acciones  $x$  o  $y$ . El proceso pasa a comportarse como  $P$  o como  $Q$  según la acción inicial. Esto se puede utilizar para introducir comportamiento no-determinístico en el modelo.

*Observación.* La elección de qué alternativa tomar no es conocida por el usuario, sino que corresponde a una decisión interna según cómo esté implementada la herramienta en uso (en el caso de la materia, sería MTSA<sup>4</sup>).

Por ejemplo, el siguiente proceso modela el comportamiento de una máquina de bebidas, que puede expender café o té dependiendo del botón que se pulse:

```
DRINKS = ( red  -> coffee -> DRINKS
          | blue -> tea    -> DRINKS
          ).
```

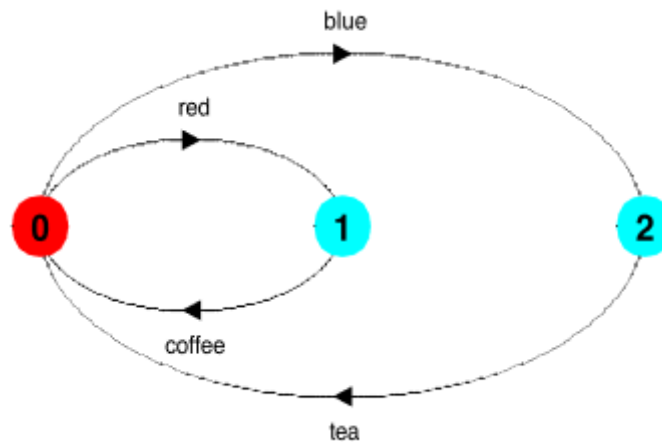


Figura 4:  $lts(DRINKS)$

*Observación.* No se permite algo de la forma  $(P \mid Q)$  sin acciones prefijando los procesos de la alternativa. Uno de los motivos de esto es garantizar que la cantidad de estados se mantenga finita.

#### 7.4.6. Composición en paralelo

Esto se usa para modelar la **conurrencia** entre procesos, representando **toda** ejecución intercalada (**interleaving**) posible entre los procesos compuestos. La idea es que la composición produce un proceso *como si fuera* secuencial, entrelanzando todas las posibles ejecuciones de los procesos compuestos.

<sup>4</sup><https://mtsa.dc.uba.ar/>

*Observación.* El modelo de concurrencia de FSP es de *interleaving*; no puede modelar la ocurrencia de dos eventos en simultáneo.

Si  $P$  y  $Q$  son procesos, entonces su composición paralela  $P \parallel Q$  representa la ejecución concurrente de  $P$  y  $Q$ .

Por ejemplo, dados los siguientes procesos:

$ITCH = (\text{scratch} \rightarrow \text{STOP}).$

$\text{CONVERSE} = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}).$

Su composición paralela se escribe como:

$\parallel \text{CONVERSE\_ITCH} = (ITCH \parallel \text{CONVERSE})$

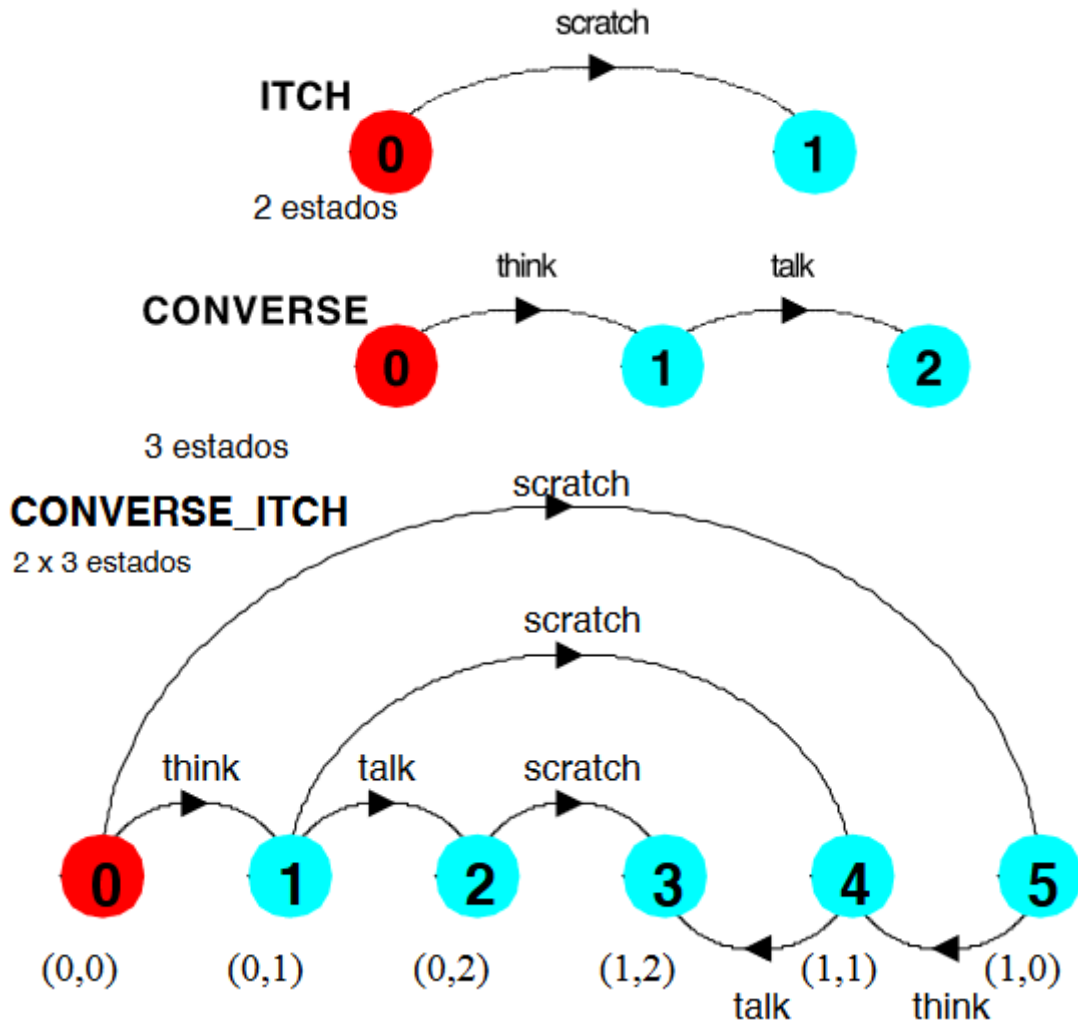


Figura 5:  $lts(ITCH)$ ,  $lts(CONVERSE)$ , y  $lts(ITCH\_CONVERSE)$ .

*Observación.* El LTS de una composición paralela de procesos corresponde a la composición paralela de los LTS de los procesos, la cual se define como se vio en 7.3.3. Es decir,

$$lts((P \parallel Q)) = lts(P) \parallel lts(Q)$$

Se puede utilizar la composición de procesos para hacer que uno de los procesos restrinja el comportamiento global del sistema, aprovechando las acciones compartidas entre ellos: una acción compartida solo puede ejecutarse en la composición si puede ejecutarse en todos los componentes que la tengan en su alfabeto.

#### 7.4.7. Elección no determinística

Un proceso de la forma  $(x \rightarrow P \mid x \rightarrow Q)$  describe una elección no determinística entre los procesos  $P$  y  $Q$ . Esto es, alternativas en las cuales la acción prefija es la misma.

El ejemplo clásico es el de la moneda:

- El siguiente proceso corresponde a una moneda no determinística:

```
COIN = (toss -> HEADS | toss -> TAILS),
HEADS = (heads -> COIN),
TAILS = (tails -> COIN).
```

- En cambio, el siguiente proceso no tiene una elección no determinística:

```
COIN = (toss -> RES),
RES = (heads -> COIN | tails -> COIN).
```

En este proceso es posible ver ambos resultados (**heads** y **tails**). En cambio, en el proceso anterior, hay un estado donde no es posible ver otro evento que no sea **heads** (idem para **tails**). Esto se puede ver claramente al animar los procesos: en este se puede elegir si sale **heads** o **tails**, mientras que en el ejemplo anterior es la herramienta la que *decide*.

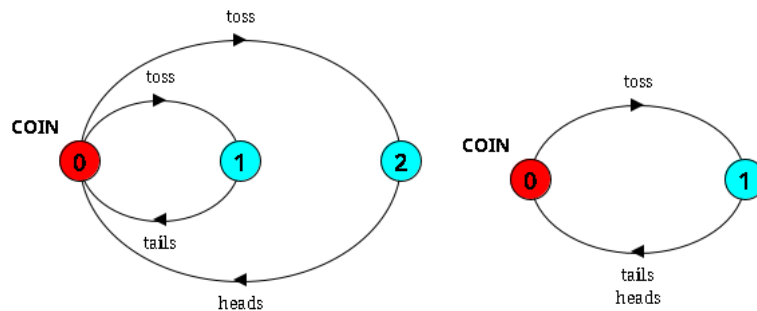


Figura 6: A la izquierda, el LTS para la moneda no determinística. A la derecha, el de la determinística.

*Observación.* Al modelar no-determinismo, no se está modelando una decisión probabilística en un espacio equiprobable, sino que el procedimiento para tomar la decisión es interno y depende de la herramienta.

### 7.4.8. Extensión del alfabetos

El alfabeto de un proceso es el conjunto de acciones en las que puede participar. En FSP se puede utilizar el operador '+' para extender el alfabeto de un proceso, agregando una o más acciones que *puede* realizar.

La clave de esto es que a pesar de que una acción agregada por extensión de alfabeto esté en el alfabeto del proceso, puede no utilizarla nunca. La diferencia entre esta extensión y no agregar nada al alfabeto se ve en la composición paralela: al una acción pertenecer al alfabeto, si esta es compartida con otro proceso, al componerlos no será posible que el proceso resultante realice tal acción (ya que los dos procesos no serán capaces de sincronizarse). En cambio, si no se extendiese el alfabeto, las acciones podrían realizarse en la composición de manera no sincronizada. Esto es útil para restringir o limitar el comportamiento de un sistema.

### 7.4.9. Ocultamiento de acciones

Las acciones especiales  $\tau$  representan cómputo interno de un proceso, y la manera de usarlas en FSP es mediante el ocultamiento de acciones (operadores '\ ' para ocultar acciones, y '@' para exponer sólo las indicadas).

**Definición 7.4.** Sea el LTS  $P = \langle S, A, \Delta, s_0 \rangle$ , y conjunto de acciones  $B \subseteq Act$ . El ocultamiento de  $B$  en  $P$  (notado  $P \setminus B$ ) es el LTS  $P' = \langle S, (A \setminus B) \cup \{\tau\}, \Delta', s_0 \rangle$ , donde  $\Delta'$  es el conjunto más chico que satisface

$$\frac{(s, \ell, s') \in \Delta}{(s, \ell, s') \in \Delta'} \ell \notin B \qquad \frac{(s, \ell, s') \in \Delta}{(s, \tau, s') \in \Delta'} \ell \in B$$

El ocultamiento de acciones tiene impacto en la composición paralela: las reglas para la relación de transición (ver Definición 7.3) se restringen a etiquetas en los alfabetos de los procesos a componer. Pero si se declara  $\tau$  como una acción, no formará parte del alfabeto, con lo cual no aplica para la regla de sincronización (aunque su nombre pudiera ser compartido con acciones de otros procesos), y siempre aplican para las reglas independientes (al ser etiquetas  $\ell \notin \alpha N$ ).

La consecuencia de esto es que cada proceso que participa en una composición paralela puede llevar a cabo sus acciones  $\tau$  (cómputo interno) independientemente de los procesos con los que interactúe.

En MTSA existe la posibilidad de minimizar un LTS para eliminar las transiciones  $\tau$  obteniendo un LTS equivalente.

## 8. Semántica de FSP y LTS

Lo siguiente será centrarse en la semántica de los procesos en FSP. Para eso, se desarrollará la idea de equivalencia entre LTS.

**Definición 8.1.** Se dice que dos términos FSP son **semánticamente equivalentes** si sus LTS son semánticamente equivalentes.

El resto de esta sección se dedica a definir qué significa que dos LTS sean equivalentes.

### 8.1. Algunas propuestas para la equivalencia de LTS

El objetivo es que la semántica de LTS defina como equivalentes a dos LTS que describen el **mismo comportamiento**.

#### 8.1.1. Isomorfismo

Una idea es definir que dos LTS son equivalentes si existe un isomorfismo  $f$  entre ellos. Es decir, dados dos LTS  $M = \langle S_M, A, \Delta_M, s_0^M \rangle$  y  $N = \langle S_N, A, \Delta_N, s_0^N \rangle$ , que exista una función biyectiva  $f : S_M \rightarrow S_N$  tal que para todo par de estados  $s, s' \in S_M$  y etiqueta  $\ell \in A$ , valga que si  $s \xrightarrow{\ell} s'$ , entonces  $f(s) \xrightarrow{\ell} f(s')$ .

Esta propuesta funciona bien con las transiciones, pero el problema es que puede haber estados inalcanzables. Un LTS que tiene estados inalcanzables no será isomorfo a uno que no los tiene, aunque sus comportamientos sean idénticos.

#### 8.1.2. Isomorfismo refinado

Un refinamiento de la idea del isomorfismo sería proponer que dos LTS son semánticamente equivalentes si existe un isomorfismo  $f$  entre sus estados alcanzables, que vincule a sus estados iniciales.

Sin embargo, esto no funciona cuando hay *estados duplicados*, que no agregan nada al comportamiento. Por ejemplo, estos dos LTS tienen el mismo comportamiento, pero no son isomorfos bajo esta última definición:

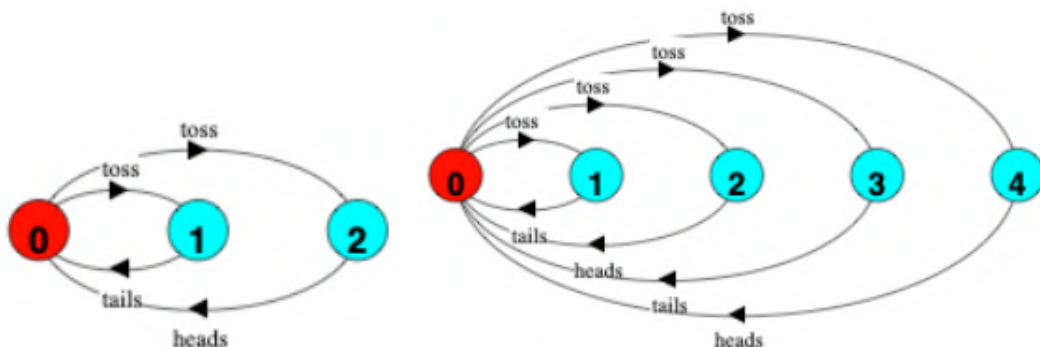


Figura 7: Estos LTS se comportan de manera idéntica, pero el de la derecha tiene más estados.

### 8.1.3. Igualdad de ejecuciones

Una idea diferente es definir que dos LTS son semánticamente equivalentes si tienen las mismas ejecuciones.

Considerar el siguiente caso:

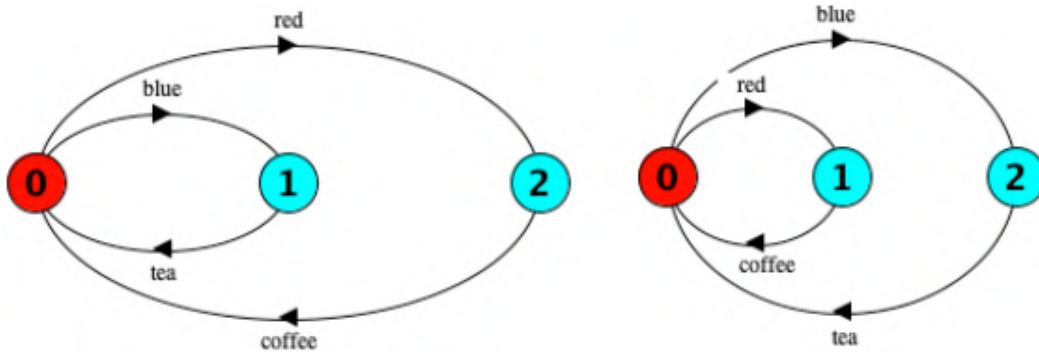


Figura 8: Estos dos LTS tienen el mismo comportamiento, pero los nombres de sus estados están *al revés*, con lo cual sus ejecuciones no son iguales: en el de la izquierda la ejecución  $0, red, 2, coffee, \dots$  es válida, y en el de la derecha vale  $0, red, 1, coffee, \dots$ .

Es decir, esta propuesta tampoco funciona, pues falla cuando estados equivalentes se llaman distinto.

### 8.1.4. Igualdad de trazas

Una variante de la idea anterior es considerar que dos LTS son semánticamente equivalentes si tienen las mismas trazas.

Sin embargo, es fácil ver que esto tampoco es adecuado: basta observar el caso de la moneda determinística y no determinística (Figura 6). Estos dos LTS tienen las mismas trazas, por lo que el criterio los definiría como equivalentes; pero claramente no lo son, ya que uno posee una elección no determinística.

## 8.2. Bisimulación

La conclusión a la que se puede llegar a partir de las propuestas fallidas para la equivalencia de LTS es que algunas son demasiado restrictivas (la igualdad de ejecuciones y el isomorfismo distinguen aspectos estructurales que son irrelevantes), y otras son demasiado laxas (la igualdad de trazas no distingue el no-determinismo).

Una semántica para LTS debería cumplir entonces:

- Ser relación de equivalencia.
- Ser abstracta respecto de la estructura.
- Ser más fina que la igualdad de trazas.
- Ser consistente con la denotación de LTS a procesos reales.
- Encajar con el lenguaje de especificación: esta es la idea de **congruencia**; una equivalencia es una congruencia con respecto a un contexto  $C(x)$  si  $P \equiv Q \Rightarrow C(P) \equiv C(Q)$ . En este



caso, el contexto es FSP, por lo que lo deseable es que si  $P$  y  $Q$  son expresiones FSP tales que  $P \equiv Q$ , entonces debería valer:

- $(a \rightarrow P) \equiv (a \rightarrow Q)$
- $(a \rightarrow P \mid b \rightarrow R) \equiv (a \rightarrow Q \mid b \rightarrow R)$
- $(P \parallel R) \equiv (Q \parallel R)$
- $P[f] \equiv Q[F]$
- $P \setminus L \equiv Q \setminus L$

Al animar un LTS en MTSA, se puede *vislumbrar* una idea sobre la equivalencia de LTS: aparecen las acciones que están habilitadas en cada momento. Lo importante entonces es qué puede hacer y qué elige hacer un LTS en cada instante, dado que lo relevante es cómo se relaciona el proceso modelado con su entorno.

Para definir esta idea formalmente, se usará el concepto de *transitar con una acción* desde un LTS a otro LTS:

**Definición 8.2.** Dado un LTS  $P = \langle S, L, \Delta, s \rangle$ , se dice que  $P$  transita con la acción  $\ell \in A$  a un LTS  $P' = \langle S, L, \Delta, s' \rangle$ , notado  $P \xrightarrow{\ell} P'$ , donde  $(s, \ell, s') \in \Delta$ .

Se usa  $P \xrightarrow{\ell}$  para decir que existe un  $P'$  tal que  $P \xrightarrow{\ell} P'$ .

Intuitivamente, se trata de pensar la transición como un cambio de un LTS a otro, que es igual pero con otro estado inicial (que es el destino de dicha transición).

### 8.3. Relaciones de bisimulación fuerte

**Definición 8.3.** Sea  $\mathcal{P}$  el universo de todos los LTS. Una relación binaria  $R \subseteq \mathcal{P} \times \mathcal{P}$  es una **bisimulación fuerte** si cuando  $(P, Q) \in R$  entonces para cada acción  $a \in Act \cup \{\tau\}$ :

- $(P \xrightarrow{a} P') \Rightarrow (\exists Q' : Q \xrightarrow{a} Q' \wedge (P', Q') \in R)$
- $(Q \xrightarrow{a} Q') \Rightarrow (\exists P' : P \xrightarrow{a} P' \wedge (P', Q') \in R)$

Con esto se puede definir cuándo dos LTS son **fuertemente bisimilares** (notado  $P \sim Q$ ). La relación de equivalencia  $\sim$  entre LTS se define como la unión de todas las relaciones de bisimulación fuerte:

$$\sim = \bigcup \{R \mid R \text{ es una bisimulación fuerte}\}$$

A su vez, se puede probar que  $\sim$  es una relación de bisimulación fuerte:

**Teorema 8.1.** Si  $P \sim Q$  entonces para cada acción  $a \in Act$ :

- $(P \xrightarrow{a} P') \Rightarrow (\exists Q' : Q \xrightarrow{a} Q' \wedge P' \sim Q')$
- $(Q \xrightarrow{a} Q') \Rightarrow (\exists P' : P \xrightarrow{a} P' \wedge P' \sim Q')$

*Demostración.* Si  $P \sim Q$ , entonces por definición de  $\sim$ , se tiene que  $(P, Q) \in R$  para alguna  $R$  bisimulación fuerte.

Luego, si  $P \xrightarrow{a} P'$ , entonces vale que  $Q \xrightarrow{a} Q'$  y  $(P', Q') \in R$  (análogamente para cuando  $Q \xrightarrow{a} Q'$ ).

Entonces, por definición de  $\sim$ , se tiene que  $(P', Q') \in \sim$ , o lo que es lo mismo,  $P' \sim Q'$ .  $\square$

La idea de la bisimulación es que dos LTS  $A$  y  $B$  son bisimilares si se pueden *simular mutuamente*, en ambas direcciones. Esto es más fuerte que decir “ $A$  puede simular a  $B$  y  $B$  puede simular a  $A$ ”, ya que  $A \sim B$  significa que cada uno puede simular al otro, pero que además en cada paso se puede elegir cuál simula a cuál.

### 8.3.1. El juego de la bisimulación fuerte

La bisimulación (fuerte) se puede reformular como un **juego** de dos jugadores:

- Un atacante, que quiere mostrar que dos LTS **no** son bisimilares.
- Un defensor, que quiere mostrar que **sí** son bisimilares.

El tablero de juego es un par de LTS, que hay que decidir si son o no fuertemente bisimilares. En cada ronda:

1. El atacante elige un LTS y lo *hace transitar* por alguna transición habilitada.
2. El defensor toma el LTS que no tomó el atacante en el turno, y debe hacerlo transitar por una transición con la misma etiqueta que el atacante.

*Observación.* El atacante puede elegir cualquiera de los dos LTS en cada turno, no tiene por qué mover el mismo en todo el juego.

Si el defensor no puede jugar (no puede *simular* la transición hecha por el atacante), entonces el juego termina, y gana el atacante (los LTS **no** son bisimilares): el que no puede hacer su movimiento, pierde. Por otro lado, si el defensor nunca pierde, entonces gana (los LTS **sí** son bisimilares).

Dicho de otra forma, si  $P$  y  $Q$  son los LTS del juego, se tiene que  $P \sim Q$  si el defensor tiene una **estrategia ganadora universal** partiendo de  $P, Q$ ; además, las configuraciones visitadas en el juego representan una relación de bisimulación. Contrariamente,  $P \not\sim Q$  si el atacante tiene una estrategia ganadora universal.

*Observación.* Esto supone que ambos jugadores son “perfectos”, es decir que siempre hacen la jugada óptima, y si existe una estrategia ganadora, la encontrarán.

Este juego es útil para probar que dos LTS **no** son bisimilares; para el caso afirmativo, es más directo dar una relación de bisimulación (no es tan interesante el juego en ese caso, dado que será infinito).

### 8.3.2. Análisis de bisimulación

El algoritmo para decidir si  $P \sim Q$  es un **algoritmo de punto fijo**: empieza con todos los pares de LTS posibles que pertenecen a una relación de bisimulación (esto es  $\sim_0$ ), y luego va

calculando iterativamente nuevas relaciones  $\sim_1, \sim_2, \dots$  mediante la eliminación de pares; esto se repite hasta que se estabiliza el procedimiento (i.e. cuando  $\sim_{i+1} = \sim_i$  para algún  $i$ ).

Cuando eso ocurre, el algoritmo alcanzó un punto fijo de la función computada. Dicha función se define inductivamente: en el paso  $n+1$ , se agrega un par si los LTS se pueden imitar un paso, y dar un resultado que pertenezca al conjunto anterior  $\sim_n$ .

Efectivamente, ocurre que  $\sim_i$  representa todos los pares de LTS que se pueden bisimilar por al menos  $i$  pasos (notar que  $\sim_0$  son todos los pares, pues  $P$  y  $Q$  se *simulan mutuamente* por cero pasos).

$$\begin{aligned} \sim_0 &\triangleq \mathcal{P} \times \mathcal{P} \\ P \sim_{n+1} Q &\triangleq : \\ 1. &\text{ if } P \xrightarrow{\mu} P', \text{ then there is } Q' \text{ such that } Q \xrightarrow{\mu} Q' \text{ and } P' \sim_n Q'. \\ 2. &\text{ if } Q \xrightarrow{\mu} Q', \text{ then there is } P' \text{ such that } P \xrightarrow{\mu} P' \text{ and } P' \sim_n Q'. \end{aligned}$$

Figura 9: Definición inductiva de la función que computa  $\sim_i$ .

La complejidad de este algoritmo no es buena: para llegar a  $\sim_1$  desde  $\sim_0$  se realiza una cantidad de operaciones cuadrática respecto a la cantidad de estados de  $P$  y  $Q$ . En peor caso, se elimina un par de LTS por cada paso, por lo que hay que hacer un número cuadrático de eliminaciones de pares. Sin embargo, a pesar de no ser la mejor complejidad posible, es tratable (no exponencial).

*Observación.* Si en el algoritmo se elimina el par de estados iniciales, significa que los LTS  $P$  y  $Q$  no pueden *bisimularse* desde un principio. En conclusión,  $P \not\sim Q$ .

## 8.4. Relaciones de bisimulación débil

Mirando la Definición 8.3, se puede ver que la bisimulación fuerte contempla las acciones de cómputo interno  $\tau$  entre las acciones que los LTS deben poder *imitar*. Esto trae como consecuencia que dos LTS que *parecerían* ser equivalentes (se comportan de manera idéntica desde el punto de vista del entorno), resultan no serlo bajo la semántica de la bisimulación fuerte, cuando uno de ellos tiene acciones  $\tau$  y el otro no.

Una idea razonable es tratar las acciones de cómputo interno de manera diferenciada para que la relación de bisimilitud capture el hecho de que se trata, justamente, de acciones internas.

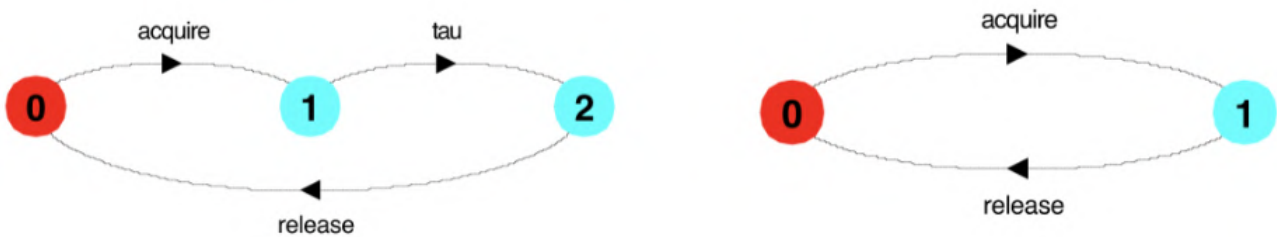
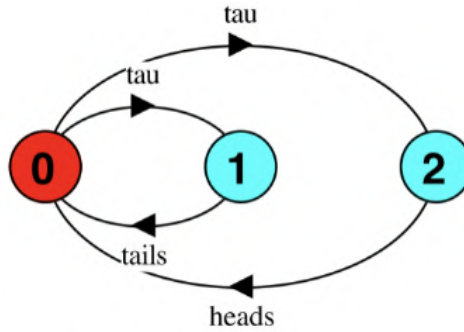


Figura 10: Estos dos LTS no son (fuertemente) bisimilares, a pesar de tener el mismo comportamiento.

Para *solucionar* esto, se puede eliminar las transiciones por acciones  $\tau$  (à la eliminación de transiciones  $\lambda$  en autómatas finitos). Sin embargo, esto podría cambiar el comportamiento de

un proceso, e.g. dejando de capturar el no-determinismo. Por ejemplo, considerar el LTS de la *moneda no determinística* de la Figura 6, al que se le oculta la acción **toss**:



Al eliminar las transiciones por acciones internas, se obtiene el siguiente LTS, donde se ha perdido el no-determinismo:



En rigor, tiene sentido que la eliminación de transiciones  $\tau$  no tenga el efecto deseado aquí, dado que esta noción está originalmente concebida para semánticas basadas en igualdad de trazas (en el caso de autómatas finitos, igualdad de lenguajes reconocidos), y previamente se vio que esto es insuficiente para capturar la semántica de los LTS.

Una solución más adecuada es definir el concepto de **bisimulación débil**, que combina la bisimulación con la clausura transitiva por transiciones  $\tau$ .

Para eso, se define la relación de *transitar con dos líneas* ( $\xRightarrow{a}$ ) de la siguiente forma:

$$\xRightarrow{a} = \begin{cases} \left( \left( \xrightarrow{\tau} \right)^* \circ \xrightarrow{a} \circ \left( \xrightarrow{\tau} \right)^* \right) & \text{si } a \neq \tau, \\ \left( \xrightarrow{\tau} \right)^* & \text{si } a = \tau, \end{cases}$$

La idea es que, para imitar una acción que no sea interna, se permite usar cualquier cantidad de transiciones  $\tau$  antes y después de hacer dicha acción. Y para imitar una acción interna, se puede hacer cualquier cantidad de acciones internas  $\tau$  (incluyendo ninguna, es decir *quedarse quieto*).

Con esto, se puede definir la bisimulación débil, que es igual que la fuerte, pero se reemplazan las flechas simples ( $\rightarrow$ ) por flechas dobles ( $\Rightarrow$ ) en el consecuente de la definición:

**Definición 8.4.** Sea  $\mathcal{P}$  el universo de todos los LTS. Una relación binaria  $R \subseteq \mathcal{P} \times \mathcal{P}$  es una **bisimulación débil** si cuando  $(P, Q) \in R$  entonces para cada acción  $a \in Act \cup \{\tau\}$ :

- $(P \xrightarrow{a} P') \Rightarrow (\exists Q' : Q \xRightarrow{a} Q' \wedge (P', Q') \in R)$
- $(Q \xrightarrow{a} Q') \Rightarrow (\exists P' : P \xRightarrow{a} P' \wedge (P', Q') \in R)$

Ahora sí, se define cuando dos LTS son débilmente bisimilares (notado  $\approx$ ) de la siguiente manera:

$$\approx = \bigcup \{R \mid R \text{ es una bisimulación débil}\}$$

*Observación.* La relación  $\approx$  es, a su vez, una bisimulación débil. Esto se puede probar de manera equivalente a lo visto en el Teorema 8.1.

Intuitivamente, la bisimulación débil considera que las acciones  $\tau$  son distintas de las otras acciones: como lo que importa a la hora de analizar procesos es el comportamiento observable, se admite que exista cómputo interno antes y después de cada acción observable sin romper bisimilitud, siempre y cuando en algún momento se pueda comportar visiblemente como el LTS con el que se esté comparando.

**Ejemplo 8.1.** Los siguientes LTS  $P$  y  $Q$  son débilmente bisimilares:

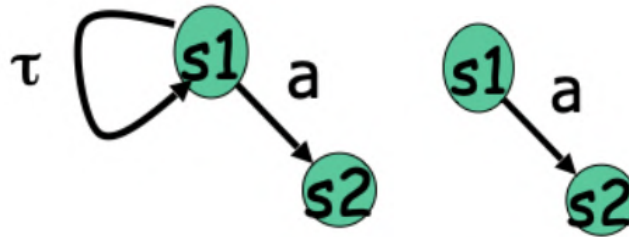


Figura 11: A la izquierda  $P$ , a la derecha  $Q$ .

Observar que  $P$  podría hacer cómputo interno infinitamente mientras que  $Q$  sólo puede hacer  $a$ . Esto podría sugerir que  $P \not\approx Q$ , pero también es cierto que en ambos casos existe, en el estado  $s_1$ , la opción de hacer  $a$ . Es por eso que  $P \approx Q$ <sup>5</sup>.

Existe también una formulación de la bisimulación débil como un juego de dos jugadores. Es el mismo juego que en el caso fuerte, con la salvedad de que el atacante hace jugadas con transiciones “reales”, mientras que el defensor puede jugar su turno con más de una transición si decide utilizar acciones  $\tau$ .

Además, hay un algoritmo con complejidad tratable que resuelve el problema de la bisimilitud débil.

**Proposición 8.1.** La relación de bisimilaridad débil  $\approx$  es más débil que la relación de bisimilaridad fuerte  $\sim$ . Es decir,

$$\sim \Rightarrow \approx$$

*No vale la vuelta.*

## 8.5. Consecuencias de la congruencia

La congruencia entre la semántica de bisimulación y FSP existe, pero se puede romper fácilmente al cambiar cosas del lenguaje. Por ejemplo, si se permitieran alternativas sin acciones prefijos (es decir, un término de la forma  $(P \mid Q)$  sería válido), entonces se rompería la congruencia.

<sup>5</sup>Esta no es la única manera de definir la bisimulación. De hecho, es debatible y existen otras, pero así es como se estudia en la materia.

**Ejemplo 8.2.** Sean los siguientes procesos:

$$Q = (a \rightarrow \text{STOP}).$$

$$P1 = (c \rightarrow \text{STOP}).$$

$$P2 = (b \rightarrow c \rightarrow \text{STOP}) \setminus \{b\}.$$

Se tiene que  $P1 \approx P2$ , y sin embargo  $(P1 \mid Q) \not\approx (P2 \mid Q)$ .

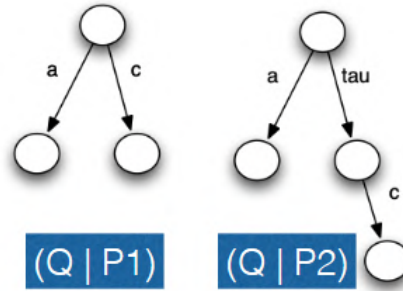
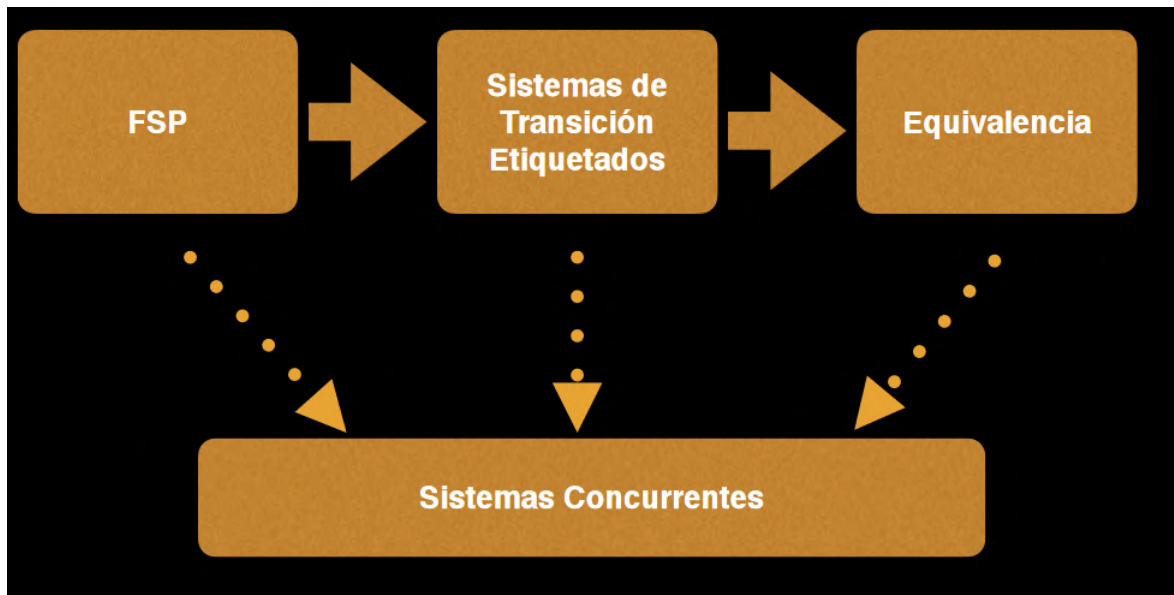


Figura 12: LTS del ejemplo

El establecimiento de congruencia entre FSP, LTS y la equivalencia de LTS vía bisimulación posibilita el **razonamiento formal y automático** acerca de sistemas concurrentes, permitiendo desarrollar este razonamiento para FSP, LTS o clases de equivalencia de LTS de manera intercambiable, sin alterar el significado. Esto da la garantía de que se opera correctamente sobre la interpretación de los sistemas concurrentes.



Por otro lado, existe la posibilidad de usar la **minimización** de LTS: dado un LTS  $P$ , obtener un LTS  $Q$  con la mínima cantidad de estados posibles y tal que  $P \sim Q$  (o  $P \approx Q$ , dependiendo de la minimización usada<sup>6</sup>). Esto puede ser útil para reducir un LTS con cálculos internos a su *mínima expresión*, para luego hacer una validación de que su comportamiento sea el esperado, a partir de analizar su bisimilaridad respecto de un LTS que sólo realice las acciones observables.

<sup>6</sup>Una manera de minimizar un LTS es calcular la relación de bisimilitud entre el LTS y sí mismo.

## 9. Análisis de Sistemas Concurrentes

El objetivo ahora es apoyarse en los modelos de sistemas concurrentes (FSP, LTS) para responder preguntas sobre ellos: esto es el **análisis** de los sistemas concurrentes, que consta de dos principales componentes:

1. Validación: *validar* el sistema contra la realidad (actual o futura, hipotética según lo que se espera del sistema en cuanto a su comportamiento; es decir, en base a un *modelo mental* sobre el problema a resolver).
2. Verificación: *vincular* dos modelos, comprobando que estén relacionados formalmente (e.g. equivalencia).

### 9.1. Validación

Es un proceso cuyo objetivo es incrementar la confianza de que un modelo formal se corresponde con la realidad: **contrastar** el modelo con un **modelo mental** del problema o solución. Es un problema que indefectiblemente requiere trabajo manual (pericia, lucidez, creatividad).

Para poder llevar a cabo el proceso de validación, es importante que el lenguaje de modelado sea *bueno*. Esto es, que el modelo tenga semántica precisa, y sea manipulable de manera tal que se preserve la semántica y la denotación (que no cambie el significado al pasar por el lenguaje y manipular los términos). Aprovechando esto, se puede usar herramientas matemáticas para disponer de distintas *vistas* del modelo formal.

Estas vistas diferentes permiten a la persona que esté haciendo la validación realizar una **inspección formal** y tener mayores chances de encontrar errores, al poder visualizar el modelo de distintas maneras.

En el caso de los modelos para sistemas concurrentes, efectivamente se pueden visualizar de distintas maneras preservando semántica y denotación: código FSP (que se puede equipar con funcionalidades de *pretty printing*), autómatas LTS (que pueden ser animados, minimizados...).

### 9.2. Verificación

Es un proceso cuyo objetivo es garantizar que dos modelos formales están **vinculados** mediante una relación formal. Por ejemplo, decidir si existe una equivalencia entre dos conjuntos de fórmulas lógicas sería un caso de verificación.

Es un proceso que se puede hacer de forma manual, semi-manual o automática. Sin embargo, va a requerir *participación humana* al momento de validar que las definiciones matemáticas usadas sean correctas y adecuadas. Siempre existe un “salto de fe” entre el modelo mental que tiene quien lo ideó, y la fórmula de especificación formal escrita.

En consecuencia, aun automatizando en un 100 % la verificación, no se puede evitar la validación, que necesariamente tendrá un componente manual.

Para el caso de sistemas concurrentes representados con el lenguaje FSP, hay diferentes estrategias de verificación:

- La **bisimulación** permite verificar equivalencia entre modelos a partir de la especificación abstracta y el ocultamiento de acciones en FSP.

- La **alcanzabilidad** de los estados, que se puede chequear usando conceptos de teoría de grafos sobre los LTS, permite verificar que un modelo sea libre de deadlocks y de estados de error.
- Los **observadores** permiten verificar que se cumplan ciertas propiedades expresadas en lenguaje formal.
- Las nociones de **satisfactibilidad** de fórmulas lógicas permiten verificar propiedades formales escritas en alguna lógica adecuada, como puede ser LTL (lógica temporal lineal).

### 9.3. Deadlock

En un sistema concurrente, la noción de deadlock corresponde a un estado del sistema en el cual éste no es capaz de progresar, pues dos o más procesos requieren recursos que están siendo utilizados por otro proceso, y se *bloquean esperando*.

#### 9.3.1. Detección de deadlock

En un LTS, un **estado de deadlock** es un estado sin transiciones salientes. Utilizando BFS (breadth first search) desde el estado inicial, es posible encontrar un estado de deadlock y un camino mínimo hacia él. A su vez, este camino será una traza del sistema que lleva a ese deadlock. Es por eso que conviene usar BFS y no otro algoritmo como DFS: es útil que el propio algoritmo de detección de deadlock encuentre una traza de ejemplo.

La complejidad temporal de la detección de un estado de deadlock es de  $O(|\Delta|)$ , es decir la cantidad de transiciones del LTS. Esta complejidad lineal puede ser un problema: la composición paralela de LTS produce lo que se conoce como **explosión de estados**. Esto es, en peor caso,  $|S|^c$ , siendo  $|S|$  la cantidad de estados de una componente y  $c$  la cantidad de componentes. Es decir, la cantidad de estados de la composición paralela de LTS crece **exponencialmente** respecto a la cantidad de componentes involucradas. Esto es un problema de escalabilidad.

Existen algoritmos que evitan construir todo el espacio de estados para mejorar la complejidad. Por ejemplo, la reducción de órdenes parciales: si un estado de la composición tiene 2 eventos salientes que pertenecen a componentes distintas (no son compartidos), entonces si uno sucede, el otro no podrá suceder en ese momento pero seguirá habilitado en el siguiente estado. Por lo tanto, la ejecución de ambos eventos llega al mismo resultado sin importar el orden en el que hayan sido ejecutados (no importa cuál sucede antes). Si se puede identificar estos casos, es posible ignorar toda una parte de la composición paralela, decidiendo no explorarla en la búsqueda de deadlock, ya que ese comportamiento ya está siendo cubierto por otro *interleaving*.

#### 9.3.2. Soluciones para deadlock

Las soluciones posibles para los deadlocks en procesos FSP son las mismas que aplican para cualquier sistema concurrente:

- Usar **reordenamientos** de las acciones la forma en que los procesos involucrados en el deadlock obtienen los recursos compartidos.
- Aplicar **timeouts** para evitar que un proceso se quede esperando eternamente por un recurso. Notar que esto debe ser una acción de cómputo interno ( $\tau$ ), ya que no debe ser sincronizada entre procesos.



- Preemption via **watchdog**: un watchdog (*perro que mira*, i.e. perro guardián) es un proceso que observa el avance del sistema, y aborta un proceso en caso de detectar que no hay avance.
- La parametrización se puede usar para **romper simetrías**. Por ejemplo, en el clásico caso de *los filósofos que cenar*, esto sería hacer que algunos agarren primero el tenedor de la izquierda, y otros primero el de la derecha.

## 9.4. Exclusión mutua y estados de error

La exclusión mutua es necesaria para evitar **interferencias** entre procesos con recursos compartidos.

Para detectar interferencias, se puede inspeccionar el LTS (o el código FSP<sup>7</sup>), pero también se puede hacer mediante un proceso de verificación: es para eso que en FSP/LTS se hace un añadido al lenguaje de modelado, que son los **estados de error**.

*Observación.* Agregar un nuevo componente al lenguaje de modelado (estados de error) puede requerir modificar la definición de composición paralela y bisimulación adecuadamente.

Estos estados de error son estados distinguidos, que no tienen transiciones salientes (al igual que los estados de deadlock). La diferencia que tienen estos estados respecto de los de deadlock es que en una composición paralela, un estado es de error si cualquiera de las componentes compuestas está en estado de error. Es decir, en la composición paralela  $P||Q$ , si  $s_i \in S_P$  (o  $s'_j \in S_Q$ ) es un estado de error en  $P$  (resp. en  $Q$ ), entonces el estado  $(s_i, s'_j)$  será de error en la composición.

Es posible detectar estados de error de la misma manera que se hacía para estados de deadlock: usando BFS desde el estado inicial, para verificar si el estado de error es alcanzable, y dar una traza para alcanzarlo en caso afirmativo.

Esto se puede utilizar para realizar **test harnesses**: código y datos de test con el objetivo de testear un sistema estimulándolo bajo diferentes condiciones, y observando su comportamiento (el sistema va a un estado de error cuando el test falla).

Los tests restringen el comportamiento del sistema (la forma de testear será a través de composición paralela), haciendo más chico el problema de verificación. Sin embargo, los tests no ofrecen garantías totales: al ser una restricción sobre el sistema, no son exhaustivos, ya que no se testea todo posible comportamiento.

Al utilizar la herramienta MTSA, se puede verificar todos los posibles ordenamientos (*interleavings*) que sean consistentes con la restricción del test. Esto ayuda para testear sistemas concurrentes, donde puede pasar que un test “tradicional” a veces falle y a veces no, dependiendo del orden en el que se hayan ejecutado los procesos.

## 9.5. Observadores

Una mejora para los tests con el objetivo de conseguir más garantías sería escribir propiedades que sean **observadas** por algún proceso en la ejecución del sistema; de esta manera, si la propiedad se cumple, no bloquea al sistema, y si no se cumple, va a un estado de error.

La gracia es que el proceso observador no restringe el comportamiento del sistema (mientras la propiedad siga valiendo, no hace nada; de hecho, si el proceso observado cumple siempre la

---

<sup>7</sup>Si se desea sufrir.

propiedad del observador, entonces la composición entre ellos será bisimilar respecto del proceso original).

Esto permite ofrecer garantías totales. La complicación está en que no suele ser fácil darse cuenta de si el supuesto observador puede o no bloquear al sistema mientras la propiedad se cumple.

### 9.5.1. Observadores en FSP

En FSP, existe la palabra clave **property** para especificación de observadores. La manera de hacerlo es escribir el *comportamiento aceptable* a continuación de la palabra clave.

El efecto al componer el proceso a observar con la propiedad es que se agregan transiciones al estado de error para cada transición que no esté habilitada en algún estado según lo especificado por la propiedad. Es decir, en todos los estados, todas las acciones del alfabeto de la propiedad están habilitadas, y ninguna más.

*Observación.* Esto último implica que si se extiende el alfabeto de la propiedad usando el operador '+', esa acción llevará a un estado de error siempre, dado que pertenece al alfabeto de la propiedad pero no es parte del *comportamiento aceptable*.

Esto garantiza que la composición del proceso y la propiedad no puede bloquear el sistema.

*Observación.* El *comportamiento aceptable* especificado por una propiedad debe ser **determinístico**.

## 9.6. Safety y Liveness

Hay dos tipos principales de propiedades para sistemas concurrentes (además del deadlock, y asumiendo que el sistema es libre de deadlock):

1. **Safety**: “nunca nada malo sucede”, es decir que el sistema es *seguro*. Todos los contraejemplos son trazas finitas.
2. **Liveness**: “algo bueno siempre sucede”, es decir que el sistema llega a buen puerto eventualmente (e.g. un livelock no es un deadlock y tampoco viola safety, pero sí liveness, pues no hay progreso en el sistema). Todos sus contraejemplos (salvo deadlock) son trazas infinitas.

### 9.6.1. Composicionalidad de safety

Las propiedades de safety se dicen **composicionales**: si  $S$  satisface una propiedad de safety  $P$ , entonces  $S||T$  también satisface  $P$ .

La idea de esto es que  $T$  restringe el comportamiento de  $S$ , por lo que si en  $S$  “no ocurría nada malo”, entonces menos aún podrá ocurrir en una versión más restringida como  $S||T$ .

A su vez, si  $P$  es un autómata observador, y no hay un estado de error alcanzable en  $S||P$ , entonces tampoco se puede alcanzar un error en  $(S||Q) || P$ .

### 9.6.2. Progreso en FSP

El progreso es una propiedad de liveness. Intuitivamente, es lo opuesto a la noción de *inanición* (starvation, cuando a un proceso se le deniegan los recursos que necesita infinitamente), referido a acciones que denotan un progreso útil (e.g. usar un recurso, no simplemente intentar obtenerlo).

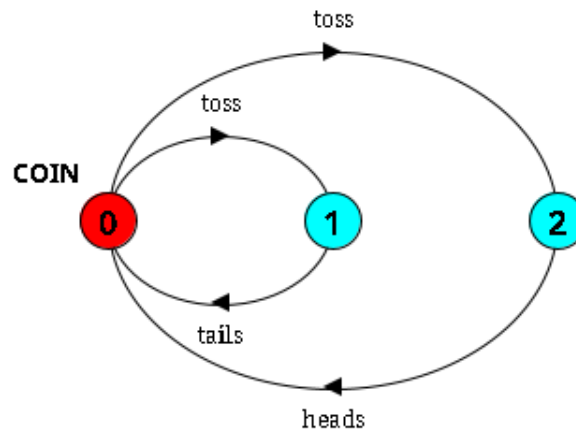
En MTSA, se puede especificar propiedades de liveness usando la palabra clave **progress**, con la siguiente sintaxis y semántica:

- Una propiedad **progress**  $P = \{a_1, a_2, \dots, a_n\}$  es verdadera si para cada ejecución infinita, por lo menos una acción entre  $a_1, a_2, \dots, a_n$  aparece infinitas veces, bajo la suposición de **fair choice**.

*Observación.* En MTSA, esto requiere un modelo sin deadlocks.

La condición de **fair choice** significa fairness sobre la manera en la que un scheduler elige las transiciones que se van ejecutando en un LTS. Una ejecución de un LTS es válida bajo fair choice si cuando un estado es visitado infinitas veces, entonces todas sus transiciones salientes también lo son.

**Ejemplo 9.1.** Considerar el modelo de la moneda no determinística:

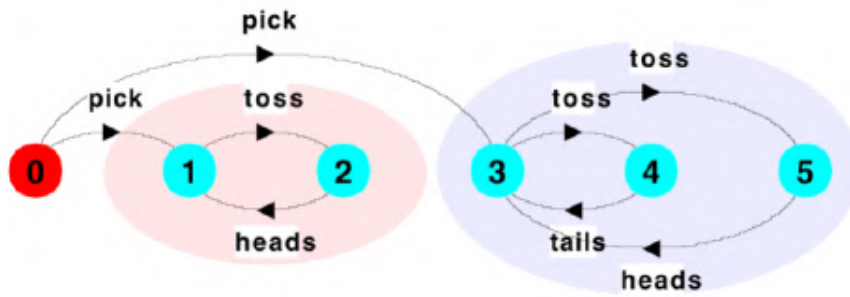


La ejecución  $(\text{toss}, \text{heads})^\omega = \text{toss}, \text{heads}, \text{toss}, \text{heads}, \dots$  no es válida bajo fair choice, pues la transición **toss** que va desde el estado 2 hacia el estado 0 no se ejecuta infinitas veces, pero el estado 0 sí se visita infinitas veces.

Las siguientes propiedades de progreso se cumplen:

- **progress HEADS** = {heads}
- **progress TAILS** = {tails}

**Ejemplo 9.2.** Considerar el LTS para una *moneda trucha*, que puede comportarse como una moneda *normal*, o bien siempre resultar en **heads**:



Este modelo cumple las propiedades:

- $\text{progress HEADS} = \{\text{heads}\}$ , ya que sin importar a dónde se vaya por `pick`, la transición `heads` se ejecuta infinitas veces.
- $\text{progress HEADSORTAILS} = \{\text{heads}, \text{tails}\}$ , ya que siempre se ejecuta `heads` una cantidad infinita de veces, y eso es suficiente para cumplir la propiedad.

Pero la siguiente propiedad no se cumple:

- $\text{progress TAILS} = \{\text{tails}\}$ , ya que si la acción `pick` lleva a la componente roja de la imagen, entonces la acción `tails` nunca se va a ejecutar.

En MTSA, se puede verificar si un sistema cumple una propiedad de progreso especificada. Esto es posible gracias a un algoritmo de detección de violaciones de progreso.

La idea del algoritmo es identificar los conjuntos de estados y transiciones alcanzables desde el estado inicial, y desde los cuales no es posible *salir*, pero donde todos los estados de estos conjuntos son alcanzables entre sí. En términos de teoría de grafos, esto son las **componentes fuertemente conexas terminales** (o BSCC, Bottom Strongly Connected Component). En resumen, una BSCC cumple que:

- Todo estado es alcanzable desde cualquier otro.
- No existen transiciones desde un estado del conjunto a un estado fuera del conjunto.

Las componentes roja y azul del LTS del Ejemplo 9.2 son BSCCs.

El algoritmo detecta estas componentes con una complejidad lineal en la cantidad de estados y transiciones del LTS. Luego, se verifica que cada componente contenga alguna de las acciones que aparecen en la propiedad a verificar (si contiene una, es seguro que esa se ejecutará infinitas veces en una traza infinita). Si alguna de las BSCCs no cumple esto, se ejecuta un BFS desde el estado inicial, para llegar a esa componente, y la traza obtenida será el contraejemplo para la propiedad.

*Observación.* La garantía de que una vez alcanzado un BSCC todas las transiciones son *inevitables* vale bajo fair choice.

### 9.6.3. Prioridad de acciones

Una manera de analizar si es razonable suponer que vale el fair choice es estableciendo prioridades en las acciones. En FSP, esto se puede hacer usando los operadores de prioridad

'<<' y '>>', que mantendrá las transiciones de mayor prioridad en el LTS frente a otras de menor prioridad cuando haya posibilidad de elegir entre ellas.

Más concretamente:

- $P \parallel C = (P \parallel Q) <<\{a_1, \dots, a_n\}$ : una composición en donde las acciones  $a_1, \dots, a_n$  tienen **más** prioridad que las demás (excluyendo  $\tau$ ). En cualquier elección donde existan una o más transiciones etiquetadas con  $a_1, \dots, a_n$ , las transiciones con acciones de **más** prioridad se descartan.
- $P \parallel C = (P \parallel Q) >>\{a_1, \dots, a_n\}$ : una composición en donde las acciones  $a_1, \dots, a_n$  tienen **menos** prioridad que las demás (incluyendo  $\tau$ ). En cualquier elección donde existan una o más transiciones **no** etiquetadas con  $a_1, \dots, a_n$ , las transiciones con acciones de **menos** prioridad se descartan.

**Ejemplo 9.3.** En las Figuras 14 y 15, se puede ver el efecto de usar operadores de prioridad sobre el siguiente proceso, cuyo LTS se ve en la Figura 13:

$\text{NORMAL} = (\text{work} \rightarrow \text{play} \rightarrow \text{NORMAL} \mid \text{sleep} \rightarrow \text{play} \rightarrow \text{NORMAL}).$

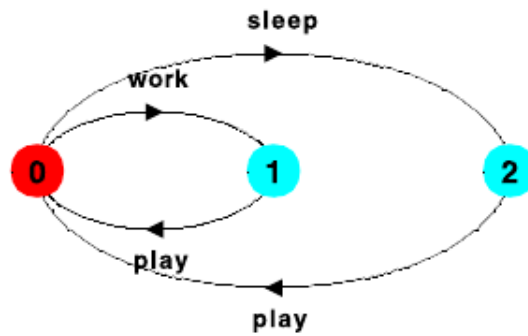


Figura 13: LTS para el proceso NORMAL.

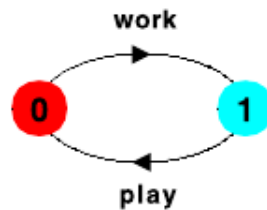


Figura 14: LTS para  $\parallel \text{HIGH} = (\text{NORMAL}) <<\{\text{work}\}$ .

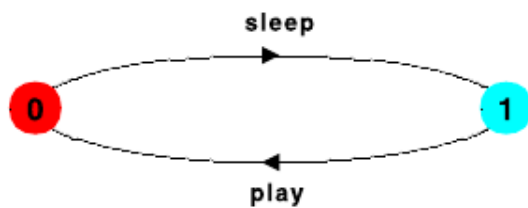


Figura 15: LTS para  $\parallel \text{LOW} = (\text{NORMAL}) >>\{\text{work}\}$ .

Usando estos operadores de prioridades, se puede forzar una situación en la que no vale fair choice.

Una manera de solucionar escenarios en los que no vale fair choice, para que pase a valer, es hacer que los procesos deban “pedir permiso” antes de efectivamente obtener un recurso compartido. Sin embargo, hay que tener cuidado con la posibilidad de un deadlock (todos los procesos están pidiendo permiso y nadie obtiene el recurso), para lo cual hay que implementar un sistema de *turnos* para poder romper la simetría.

#### 9.6.4. Composicionalidad de liveness

En el caso de las propiedades de liveness, **no** es cierto que si  $S$  satisface una propiedad  $P$ , entonces  $S||T$  deba satisfacerla también. Es decir, **no son composicionales**.

Por ejemplo, en el ejemplo de la moneda (6), componer el proceso con  $\text{STOP} + \{\text{heads}\}$  bloquea la ocurrencia del evento **heads**, con lo cual no hay progreso. Y sin embargo, la moneda original sí cumplía la propiedad de progreso  $\{\text{heads}\}$ , como se vio en el Ejemplo 9.1. La idea de este ejemplo es que hay que lograr que una acción que pertenezca al alfabeto de los dos procesos compuestos, no pueda ser sincronizada entre ellos. Una manera de hacerlo es con extensión de alfabetos (no es la única forma, se podría hacer consiguiendo que dos procesos no sean capaces de sincronizarse, por ejemplo haciendo que hagan las mismas acciones pero en orden contrario).

## 10. Model Checking usando LTL

### 10.1. Lógica modal proposicional

Las lógicas modales son aquellas que estudian la formalización del razonamiento que involucra *modalidades*: la manera clásica de hacer esto es extender los operadores usuales de la lógica proposicional ( $\wedge$ : conjunción,  $\vee$ : disyunción,  $\neg$ : negación, y  $\rightarrow$ : implicación) con los **operadores modales**:

- Necesidad (box): notado  $\Box$
- Posibilidad (diamond): notado  $\Diamond$

#### 10.1.1. Semántica

Las fórmulas modales se interpretan sobre **estructuras de Kripke** y **valuaciones** sobre el universo de variables proposicionales  $Pr$ .

**Definición 10.1.** Una estructura de Kripke es un par  $\langle W, R \rangle$  tal que:

- $W$  es un conjunto de mundos posibles.
- $R \subseteq W \times W$  es una relación binaria entre mundos.

Las valuaciones asignan un valor de verdad a las proposiciones según el mundo:

**Definición 10.2.** Una función de valuación  $v : Pr \times W \rightarrow \{T, F\}$  asigna a cada proposición un valor de verdad en cada mundo posible.

La semántica está dada por la relación de satisfacción  $\models$  entre fórmulas e interpretaciones:

**Definición 10.3.** Sea  $\mathcal{L}$  un lenguaje modal,  $\langle W, R \rangle$  una estructura de Kripke y  $v$  una función de valuación para  $\mathcal{L}$ . La *valuación* de las fórmulas de  $\mathcal{L}$  se define recursivamente como:

- $w \models p$  sii  $v(p, w) = T$ , donde  $p$  es una variable proposicional
- $w \models \neg\alpha$  sii  $w \not\models \alpha$
- $w \models \alpha \wedge \beta$  sii  $w \models \alpha$  y  $w \models \beta$
- $w \models \Box\alpha$  sii para todo mundo  $w'$  tal que  $wRw'$  sucede que  $w' \models \alpha$

Además, se dice que  $w \models \alpha$  sii  $w \models \alpha$  para todo mundo  $w \in W$ .

**Ejemplo 10.1.** 1.  $\Box(p \rightarrow q)$ : “en todos los mundos a los que se puede llegar, vale que  $p \rightarrow q$ ”.

2.  $\Box p \rightarrow \Box q$ : “si en todos los mundos alcanzables vale  $p$ , entonces en todos también vale  $q$ ”.

3.  $\Box p \rightarrow p$ : “si en todos los mundos alcanzables vale  $p$ , entonces vale  $p$  en el mundo actual”.

*Observación.* Análogamente a lo que ocurre con el cuantificador existencial  $\exists$  y el universal  $\forall$  en la lógica de primer orden, acá el operador de posibilidad  $\Diamond$  se puede definir a partir del de necesidad  $\Box$ :

- $\Diamond p$  sii  $\neg(\Box\neg p)$

El significado de  $w \models \Diamond\alpha$  es que “en el mundo  $w$ , existe al menos un estado  $w'$  donde  $wRw'$  y  $w' \models \alpha$ ”.

Como se verá después, esto se puede aplicar para la verificación automática de programas: una interpretación se puede pensar como un sistema de transición de estados, que a su vez puede representar un programa. Los estados del sistema serían los mundos de una estructura de Kripke, y las relaciones entre mundos serían las transiciones entre estados del sistema, según la ejecución del programa.

Si se tiene una propiedad  $P$  que se quiere verificar en un programa  $Prog$ , y este programa se puede representar como una estructura de Kripke  $\langle W, R \rangle$  con estado inicial  $w_0$  y  $P$  como una fórmula modal, entonces verificar que  $Prog$  cumple la propiedad  $P$  es lo mismo que comprobar que  $w_0 \models P$ . Esto es lo que se conoce como **model checking**: el procedimiento automático de verificación de un modelo contra una propiedad.

## 10.2. Lógica temporal lineal

Las lógicas temporales sirven para razonar acerca del tiempo, y pueden ser definidas como casos especiales de lógicas modales.

Se prestan a ser usadas para razonar sobre ejecuciones de un programa reactivo, por lo que tiene sentido pensar en aplicarlas para el model checking.

La primera variante de lógica temporal que se ve en la materia es la **lógica temporal lineal** (LTL), donde el tiempo se representa de manera lineal.

## 10.3. Semántica de LTL

En LTL, las transiciones entre mundos representan el paso del tiempo: una fórmula en LTL debe interpretarse sobre una estructura de Kripke  $\langle W, R \rangle$  donde  $W$  es un conjunto numerable y  $R$  una relación de orden total sobre  $W$ .

Los operadores modales se interpretan de la siguiente manera:

- $\Box P$ : “siempre en el futuro vale  $P$ ”.
- $\Diamond P$ : “en algún momento en el futuro vale  $P$ ”.

Esto da lugar a un conjunto de trazas, sobre las cuales se define la semántica: un estado (mundo) será una posición  $i$  en la traza  $\sigma$ , i.e.  $\sigma[i]$  es la  $i$ -ésima posición en la traza  $\sigma$ .



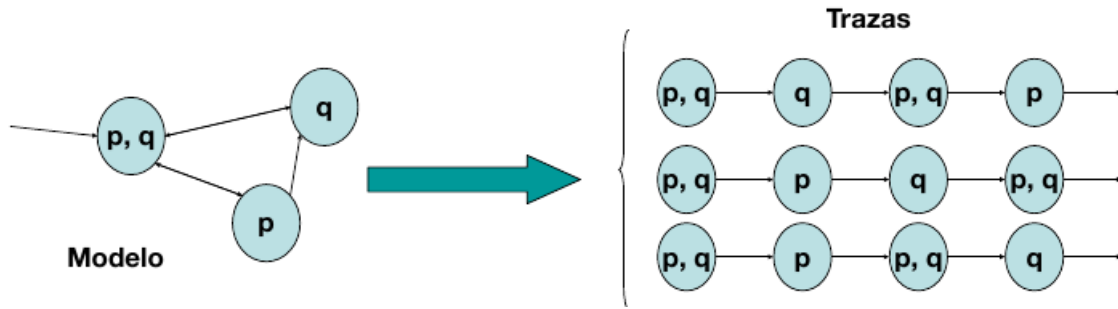


Figura 16: Acá se puede ver como el modelo representado por la estructura de Kripke se *descompone* en las posibles trazas.

Teniendo esto en cuenta, las interpretaciones de los operadores lógicos son las siguientes:

- $\sigma[i] \models p$  sii  $v(\sigma[i], p) = T$
- $\sigma[i] \models X\alpha$  sii  $\sigma[i + 1] \models \alpha$
- $\sigma[i] \models \Diamond\alpha$  sii  $\exists j : j \geq i$  y  $\sigma[j] \models \alpha$
- $\sigma[i] \models \Box\alpha$  sii  $\forall j : j \geq i$  implica  $\sigma[j] \models \alpha$
- $\sigma[i] \models \alpha \cup \beta$  sii  $\exists k : k \geq i$  y  $\sigma[k] \models \beta$  y  $(\forall j : k > j \geq i : \sigma[j] \models \alpha)$
- $\sigma[i] \models \alpha \text{ W } \beta$  sii  $\sigma[i] \models \alpha \cup \beta$  o  $\sigma[i] \models \Box\alpha$
- $\sigma[i] \models \alpha \text{ R } \beta$  sii  $\sigma[i] \models \beta \cup (\alpha \wedge \beta)$  o  $\sigma[i] \models \Box\beta$

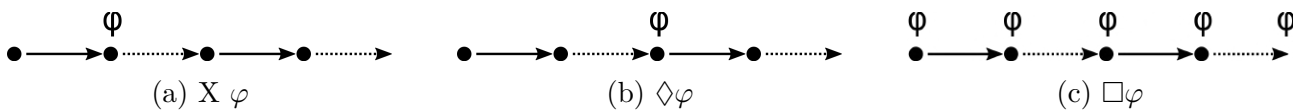


Figura 17: Diagramas que muestran en una línea de tiempo el significado de los operadores unarios de LTL.

Observación. Notar que se extiende la lógica con nuevos operadores:

- El operador  $X$  se refiere al próximo estado en el tiempo.
- El operador  $U$  es el *until*: es decir,  $P \cup Q$  significa que vale  $P$  hasta que en algún momento (que puede ser al principio) vale  $Q$ .

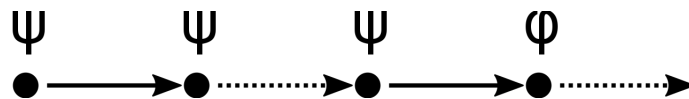
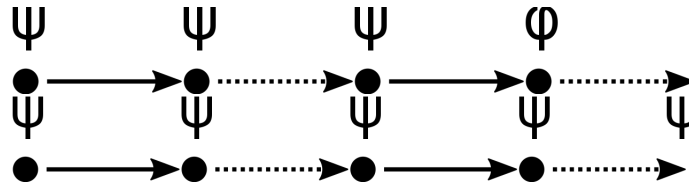
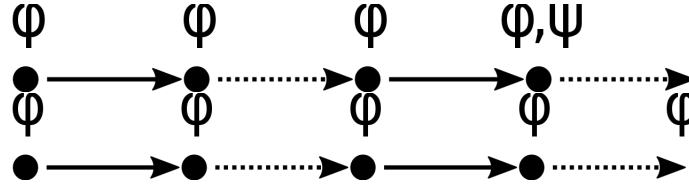


Figura 18: Diagrama para  $\psi \cup \varphi$

- El operador  $W$  es el *weak until*. Es lo mismo que el *until*, pero permite que la proposición de la derecha no valga, y que en cambio valga siempre la de la izquierda.
- El operador  $R$  es el *release*: es decir,  $P \text{ R } Q$  significa que  $Q$  debe valer hasta el instante en el que  $P$  valga por primera vez (inclusive); y si  $P$  nunca vale, entonces  $Q$  debe valer siempre.

Figura 19: Diagramas para  $\psi W \varphi$ Figura 20: Diagramas para  $\psi R \varphi$ 

Se dice que una traza verifica una propiedad si esta se cumple en el primer estado:

$$\sigma \models \alpha \text{ si } \sigma[1] \models \alpha$$

Por último, se dice que un modelo  $M$  satisface una fórmula  $\alpha$  de LTL ( $M \models \alpha$ ) si y sólo si para toda traza  $\sigma$ , vale que  $\sigma \models \alpha$ .

*Observación.* En LTL,  $PWQ$  es una propiedad de liveness, y cambiar el *until* por el *weak until* (i.e.  $PWQ$ ), hace que la propiedad sea de safety. Esto es así porque libera la necesidad de que valga  $Q$  (pide menos).

## 10.4. Vínculo entre LTL y LTS

Las estructuras de Kripke se pueden usar para modelar sistemas concurrentes (al igual que los LTS). Se puede modelar la memoria del programa, de manera consistente con sistemas reactivos, pero usando memoria compartida (en vez de comunicación por mensajes como en el modelado usado en la materia). Es decir, esta es otra manera de modelar los sistemas concurrentes: en la materia se usan los LTS y el lenguaje FSP, pero hay otras maneras.

Sin embargo, los LTS no son estructuras de Kripke: los estados de un LTS no contienen la valuación de proposiciones, por lo que las definiciones de la semántica de los operadores no tienen sentido en un LTS tal como se vieron en la materia.

Lo que se puede hacer es interpretar la traza  $t$  de un LTS como una estructura de Kripke, donde:

- El conjunto de proposiciones es el alfabeto del LTS.
- Una proposición  $P$  es verdadera en el estado  $i$  si y sólo si  $P$  es la acción en la posición  $i$  de la traza  $t$ .

Usando esto es que se desarrolla el mecanismo de model checking con propiedades LTL.

## 10.5. Algoritmo de verificación para programas concurrentes

Dado un LTS  $M$  que representa un programa concurrente, y una fórmula LTL  $P$  que representa una propiedad a verificar en el programa, el objetivo es definir un algoritmo que responda afirmativamente si **todas las trazas** de  $M$  satisfacen  $P$ .

A grandes rasgos, el algoritmo consiste en tres pasos:

1. Convertir la fórmula LTL  $\neg P$  en un autómata  $A_{\neg P}$  que caracterice a todas las trazas que satisfacen  $\neg P$ .
2. Verificar si las trazas de  $M$  y las de  $A_{\neg P}$  son disjuntas.
3. Si la respuesta a 2. es sí (i.e. la intersección entre las trazas es vacía), entonces responder afirmativamente. En caso contrario, responder negativamente y retornar una traza en la intersección como contra-ejemplo para la propiedad.

En las siguientes páginas, se verá cada parte del algoritmo en mayor detalle.

*Observación.* Notar que lo que se construye es como un observador que *avisa* cuando que se cumple la negación de la propiedad.

### 10.5.1. Autómatas de Büchi

Notar que una fórmula LTL caracteriza al conjunto de trazas que la satisface. A su vez, recordar que las trazas de los LTS son **infinitas**, pues están modelando programas reactivos concurrentes.

Por lo tanto, el autómata al que se convertirá una fórmula LTL deberá ser capaz de reconocer cadenas infinitas, para tener un lenguaje que sea el mismo que describe la fórmula LTL.

Es por eso que ninguno de los autómatas estudiados en Teoría de Lenguajes es útil para este objetivo (todos reconocen conjuntos de cadenas finitas). Se introduce entonces un nuevo tipo de autómatas: los **autómatas de Büchi**.

Un autómata de Büchi<sup>8</sup> tiene una cantidad de **estados finita**, pero reconoce conjuntos de **cadenas infinitas**. Específicamente, reconoce lenguajes  $\omega$ -regulares. Es por esto que las fórmulas LTL pueden ser traducidas a autómatas de Büchi, que aceptará una traza si y sólo si esta satisface la fórmula.

**Definición 10.4.** Un autómata de Büchi es una 5-upla  $A = \langle \Sigma, Q, \Delta, Q_0, F \rangle$ , donde:

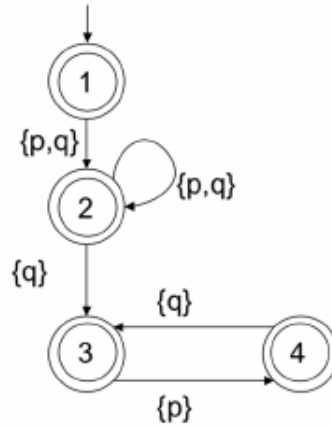
- $\Sigma$  es un alfabeto finito.
- $Q$  es un conjunto finito de estados.
- $\Delta \subseteq Q \times \Sigma \times Q$  es la relación de transición entre estados.
- $Q_0 \subseteq Q$  es el conjunto de estados iniciales.
- $F \subseteq Q$  es el conjunto de estados de aceptación.

*Observación.* La Definición 10.4 no hace referencia a determinismo o no-determinismo. Esto es porque esto no está restringido: los autómatas de Büchi pueden ser no-determinísticos.

Un autómata de Büchi **acepta** una cadena cuando la ejecución en el autómata de la misma *visita un estado de aceptación infinitas veces*.

**Ejemplo 10.2.** El autómata  $A = \langle \{p, q\}, \{1, 2, 3, 4\}, \Delta, \{1\}, \{1, 2, 3, 4\} \rangle$  se puede representar con el siguiente diagrama:

<sup>8</sup>Por el matemático J. R. Büchi: [https://en.wikipedia.org/wiki/Julius\\_Richard\\_B%C3%BCchi](https://en.wikipedia.org/wiki/Julius_Richard_B%C3%BCchi)



El lenguaje aceptado por un autómata de Büchi consiste en un conjunto de secuencias infinitas sobre su alfabeto. Siendo  $A$  un autómata de Büchi con alfabeto  $\Sigma$ , el conjunto de todas las secuencias infinitas sobre  $\Sigma$  se nota  $\Sigma^\omega$ , y el lenguaje aceptado por  $A$  se nota  $\mathcal{L}(A) \subseteq \Sigma^\omega$ .

**Definición 10.5.** Dada una secuencia infinita  $w = a_0a_1a_2... \in \Sigma^\omega$ , una **ejecución** del autómata de Büchi  $A = \langle \Sigma, Q, \Delta, Q_0, F \rangle$  sobre  $w$  es una secuencia de estados  $r = q_0q_1q_2...$ , donde  $q_0 \in Q_0$  y para cada  $i \geq 0$ , vale que  $\langle q_i, a_i, q_{i+1} \rangle \in \Delta$ .

**Definición 10.6.** Dada una ejecución  $r$ , se define como  $\text{inf}(r) \subseteq Q$  al conjunto de estados del autómata que aparecen en  $r$  una cantidad infinita de veces.

**Definición 10.7.** Una ejecución  $r$  se dice **aceptada** por un autómata de Büchi  $A = \langle \Sigma, Q, \Delta, Q_0, F \rangle$  cuando  $\text{inf}(r) \cap F \neq \emptyset$ .

Como se mencionó antes, intuitivamente  $r$  es aceptada si atraviesa al menos un estado de aceptación una cantidad infinita de veces.

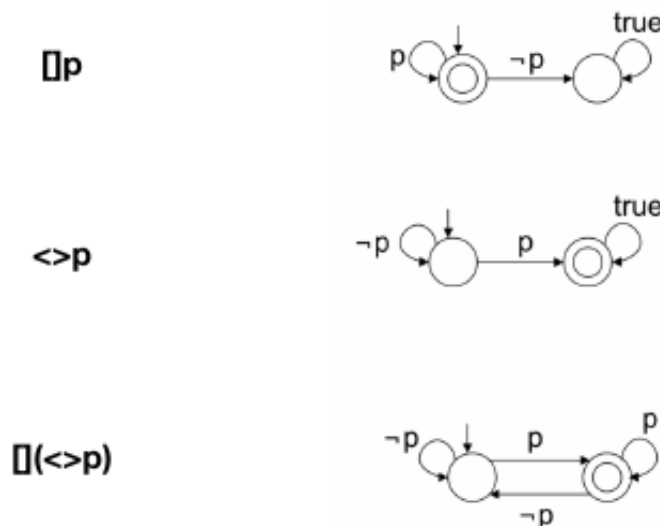


Figura 21: Ejemplo de autómatas de Büchi *equivalentes* a algunas fórmulas LTL.

*Observación.* El tamaño de un autómata de Büchi que caracteriza una fórmula LTL crece exponencialmente con respecto al tamaño de dicha fórmula.

### 10.5.2. Autómatas de Büchi generalizados

Un autómata de Büchi generalizado es lo mismo que uno convencional, salvo por el conjunto de estados de aceptación, que en este caso es un **conjunto de conjuntos de estados de aceptación**.

Formalmente:

**Definición 10.8.** Un autómata de Büchi generalizado es una 5-upla  $A = \langle \Sigma, Q, \Delta, Q_0, F \rangle$ , donde:

- $\Sigma$  es un alfabeto finito.
- $Q$  es un conjunto finito de estados.
- $\Delta \subseteq Q \times \Sigma \times Q$  es la relación de transición entre estados.
- $Q_0 \subseteq Q$  es el conjunto de estados iniciales.
- $F \subseteq \mathcal{P}(Q)$  es el conjunto de conjuntos de estados de aceptación.

Una ejecución  $r$  es aceptada por un autómata de Büchi generalizado si y sólo si  $\inf(r) \cap F_i \neq \emptyset$  para todo  $F_i \in F$ .

Intuitivamente, la ejecución se acepta si atraviesa al menos un estado de aceptación *de cada conjunto de estados de aceptación* una cantidad infinita de veces.

### 10.5.3. Conversiones entre autómatas de Büchi generalizados y no generalizados

Un autómata de Büchi  $A = \langle \Sigma, Q, \Delta, Q_0, F \rangle$  se puede convertir trivialmente en un autómata de Büchi generalizado:  $A' = \langle \Sigma, Q, \Delta, Q_0, \{F\} \rangle$ .

La conversión en sentido opuesto no es trivial: sea  $GA = \langle \Sigma, Q, \Delta, Q_0, \{F_1, \dots, F_k\} \rangle$  un autómata de Büchi generalizado. Se puede construir un autómata de Büchi *común*  $A$  del siguiente modo:

- $A = \langle \Sigma, Q \times \{1, \dots, k\}, \Delta', (Q_0, 1), F_1 \times \{1\} \rangle$
- $((q, x), a, (q', y)) \in \Delta'$  si y sólo si:
  - $(q, a, q') \in \Delta$
  - $q \in F_x \wedge x < k \Rightarrow y = x + 1$
  - $q \in F_x \wedge x = k \Rightarrow y = 1$
  - $q \notin F_x \wedge x = y$

La idea para la construcción de  $A$  es hacer  $|F|$  copias de  $GA$ , y colocar en cada copia todas las transiciones normales que tiene  $GA$ , salvo en los casos de las transiciones que salgan de estados de aceptación. Para esos casos, hacer que la transición vaya a la siguiente copia, en el estado que corresponda. En caso de estar en la última copia, hacer que vaya a la primera.

Entonces, recordando que una ejecución es aceptada por  $GA$  si se atraviesa infinitas veces al menos un estado de cada conjunto de estados de aceptación, se tiene que en  $A$  hay que pasar infinitas veces por al menos un estado de aceptación de la primera capa (por definición de  $A$ ).

La única manera de hacer esto es hacer el recorrido de todas las capas infinitas veces (por construcción de las transiciones de  $A$ ), lo que es lo mismo que pasar por al menos un estado de aceptación en cada capa (dado que es la única manera de *cambiar de capa*) infinitas veces. Se puede ver intuitivamente que esto es lo mismo que el criterio de aceptación para  $GA$ .

Además, se puede probar que  $\mathcal{L}(GA) = \mathcal{L}(A)$ .

#### 10.5.4. Conversión de LTL a autómatas de Büchi

El algoritmo para construir un autómata de Büchi para reconocer la trazas que cumplen una fórmula LTL que se ve en la materia se llama **LTL2Büchi**, y es un método de construcción basado en el método del tableau.

El método del tableau para lógica proposicional es un procedimiento para decidir si una fórmula es satisfactible. La idea es descomponer la fórmula en sub-fórmulas de manera top-down, de forma tal que se arme un árbol. Una rama *se cierra* cuando hay una contradicción, con lo cual una fórmula es satisfactible si una rama del árbol *no se cierra*.

Aplicado a LTL, el objetivo es conseguir un autómata de Büchi que acepta el mismo lenguaje que una fórmula LTL dada. Cada estado del autómata *sabr*á qué fórmula debe reconocer, y al avanzar de un estado  $s$  a un  $s'$  por una transición  $a$ , la fórmula que debe reconocer  $s'$  debería ser como la de  $s$  después de haber procesado  $a$ . Esta parte es similar a la idea de descomposición del tableau para lógica proposicional: termina cuando *se cierran* las transiciones, y eso ocurre cuando se *colapsan* dos estados con **New** vacío y donde coinciden **Old** y **Next** (ver algoritmo a continuación).

##### Algoritmo LTL2Büchi [Gerth, Peled, Vardi, Wolper; 1995]

- Entrada: fórmula LTL  $P$  en forma normal positiva (sólo las proposiciones pueden estar negadas):
  - $\neg(\alpha \text{ U } \beta) \equiv \neg\alpha \text{ R } \neg\beta$
  - $\neg(\alpha \text{ R } \beta) \equiv \neg\alpha \text{ U } \neg\beta$
  - $\neg(\text{X } \alpha) \equiv \text{X } \neg\alpha$
  - $\Diamond\alpha \equiv \text{true U } \alpha$
  - $\Box\alpha \equiv \text{false R } \alpha$
- Salida: autómata de Büchi que reconoce el mismo lenguaje que la fórmula.
- Cada estado del autómata tendrá tres conjuntos de propiedades:
  - **New**: las propiedades que deben valer desde el estado, pero que no fueron “procesadas” por el algoritmo.
  - **Old**: las propiedades que deben valer desde el estado, y que ya fueron “procesadas” (*expandidas*) por el algoritmo.
  - **Next**: las propiedades que deben valer en los estados sucesores inmediatos.
- Cada estado tendrá una lista de estados llamada
  - **Incoming**: los estados predecesores inmediatos.
- Procedimiento:

1. Crear un nodo  $n = \langle \text{New} = \{P\}, \text{Old} = \emptyset, \text{Next} = \emptyset, \text{Incoming} = \emptyset \rangle$
  2. Para cada nodo  $n$  con  $f \in \text{New}$ , procesar  $f$  creando nuevos nodos. Continuar hasta que no exista  $f \in \text{New}$  en ningún nodo  $n$ .
  3. Construir un autómata de Büchi generalizado  $GA = \langle \Sigma, Q, \Delta, Q_0, F \rangle$  a partir del obtenido, donde:
    - $\Sigma$  son subconjuntos de proposiciones de la fórmula LTL  $P$ .
    - $Q = \text{NodeList} \cup \{\text{init}\}$
    - $Q_0 = \{\text{init}\}$
    - $\Delta$  se define como:
      - $(q, d, q') \in \Delta$  sii  $q \in \text{Incoming}(q')$  y  $d$  satisface la conjunción de las proposiciones negadas y no negadas que están en  $\text{Old}(q')$
    - $F \subseteq \mathcal{P}(Q)$ , i.e.  $F = \{F_1, F_2, \dots\}$  se define como:
      - Para cada sub-fórmula  $h \text{ U } k$ , existe un estado de aceptación  $F_i$  que contiene todos los estados  $q$  tales que o bien  $k \in \text{Old}(q)$ , o bien  $h \text{ U } k \notin \text{Old}(q)$
      - Si no hay sub-fórmulas de la forma  $h \text{ U } k$ , entonces  $F = \{Q\}$ .
    - Finalmente, se puede cambiar los conjuntos de valuaciones en una transición por una fórmula proposicional que los caracterice, obteniendo ahora sí el autómata de Büchi generalizado.
  4. Transformar el autómata de Büchi generalizado en uno común (como se vio antes).
- Complejidad temporal: exponencial respecto al tamaño de la fórmula  $P$ .

A continuación, se muestran (robo descarado mediante) las diapositivas que muestran el algoritmo con más detalle y ejemplos, tal cual como está en la clase teórica<sup>9</sup>:

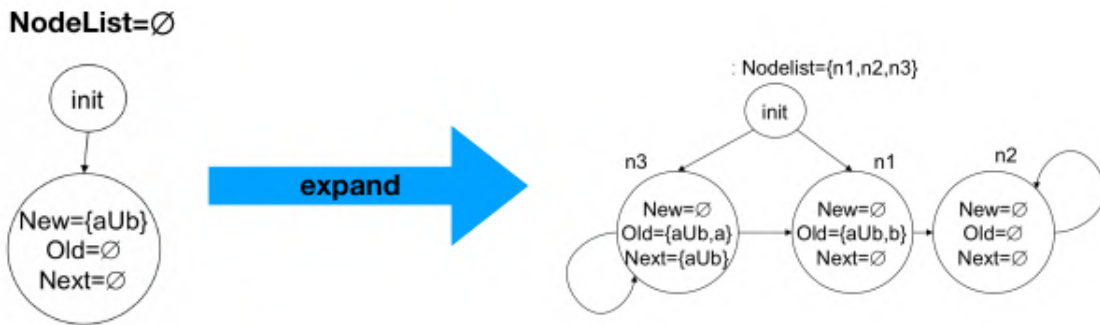


Figura 22: Ejemplo de la expansión para  $a \text{ U } b$ .

<sup>9</sup>Me niego rotundamente a transcribir esto a mano.

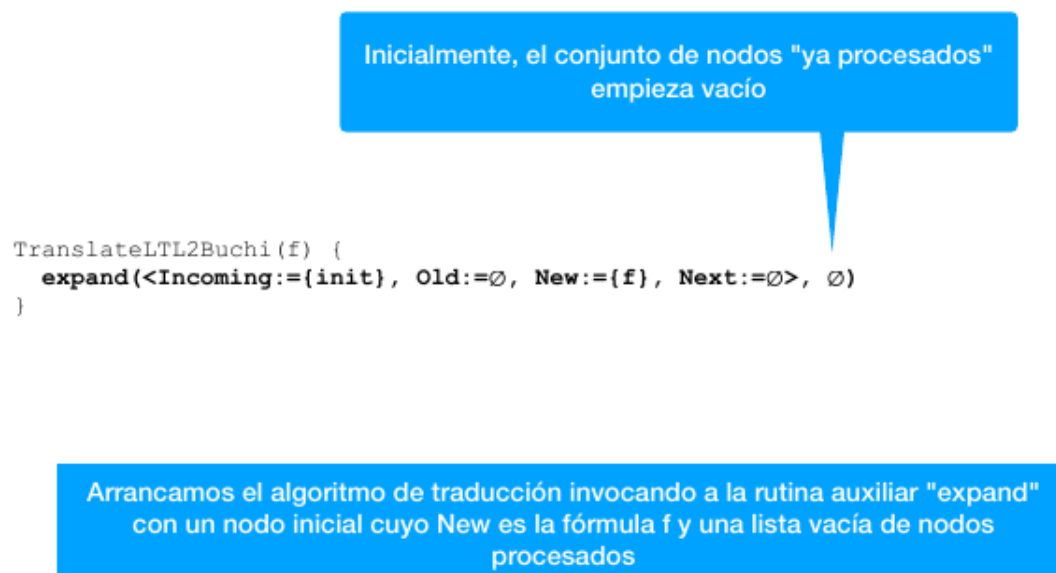


Figura 23: Principio del algoritmo.



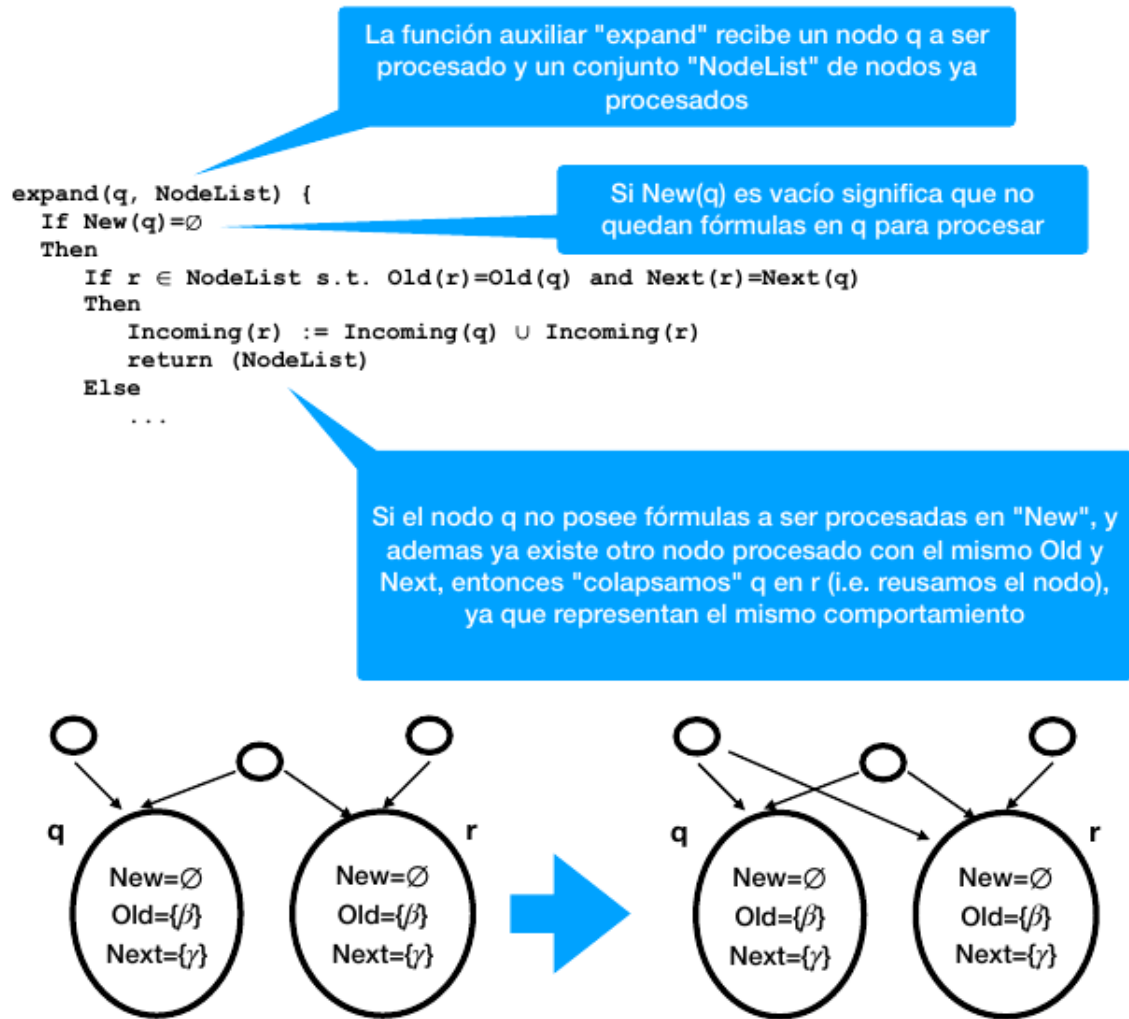


Figura 24: Caso reuso.

```

expand(q, NodeList) {
  If New(q)= $\emptyset$ 
  Then
    If  $r \in \text{NodeList}$  s.t. Old(r)=Old(q) and Next(r)=Next(q)
    Then
      Incoming(r) := Incoming(q)  $\cup$  Incoming(r)
      return (NodeList)
    Else
      Create a new nodo q' s.t.
      Incoming(q') := q
      Old(q') :=  $\emptyset$ 
      New(q') := Next(q)
      Next(q') :=  $\emptyset$ 
      Return expand(q', NodeList  $\cup$  {q})

```

Veamos ahora que pasa si no existe un nodo "r" que podamos reutilizar que ya haya sido procesado

En ese caso, creamos un nuevo nodo q' tal que "provenga" de q (el nodo que estamos procesando), marcamos para procesar las fórmulas que deben cumplirse en el siguiente estado de q (Next(q)) y marcamos como procesado a "q", llamado a expand recursivamente

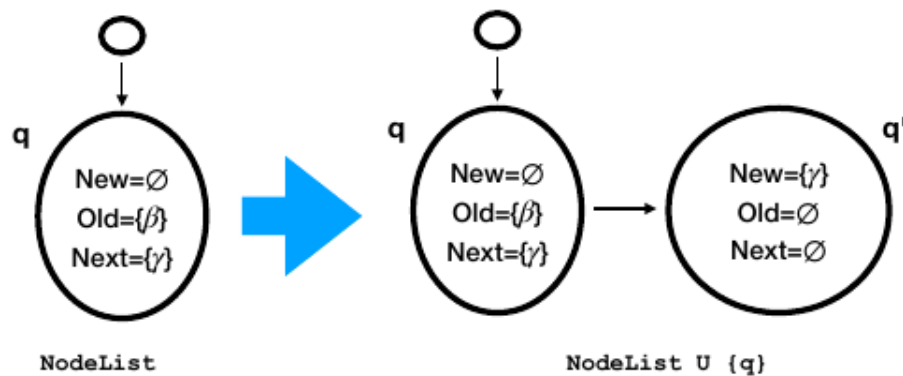


Figura 25: Caso  $\text{New} = \emptyset$  y sin reuso.

```

expand(q, NodeList) {
  If New(q)= $\emptyset$ 
  Then
    ... // procesamos si New(q) es vacío
  Else
    // New(q) no es vacío
    Pick  $f \in \text{New}(q)$ 
    New(q) := New(q) - {f}
    If  $f \in \text{Old}(q)$ 
    Then
      Return expand(q,NodeList)
    Else
      ...

```

Ahora veamos que pasa si efectivamente existen fórmulas en q que debemos procesar

Seleccionamos alguna fórmula que esté en New(q) y la eliminamos del conjunto New(q)

Si la fórmula ya esta en Old(q) (i.e. ya fue procesada), continuamos la ejecución recursivamente

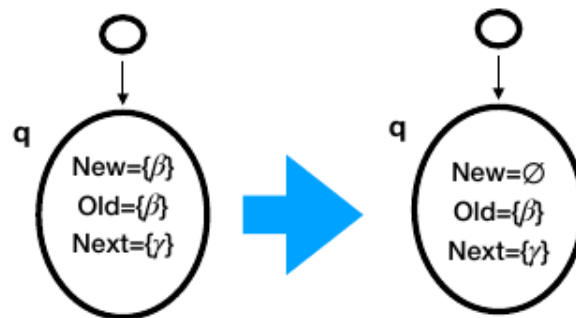


Figura 26: Caso  $\text{New} \neq \emptyset$ .

```
expand(q, NodeList) {  
  If New(q)= $\emptyset$   
  Then  
    ... // procesamos si New(q) es vacío  
  Else  
    // New(q) no es vacío  
    Pick  $f \in \text{New}(q)$   
    New(q) := New(q) - {f}  
    If  $f \in \text{Old}(q)$   
    Then  
      Return expand(q,NodeList)  
  Else  
    ...
```

Ahora, veamos que pasa si  $f$  no ha sido todavía procesada en el nodo  $q$  por el algoritmo

Para eso, separamos en casos de acuerdo a qué forma tiene la fórmula " $f$ "

Figura 27: Caso  $f$  no procesado en el nodo. Se separa en casos según la forma de  $f$  (ver figuras siguientes).

```

If  $f$  is a boolean constant or  $f \in AP$  or  $\neg f \in AP$ 
Then
  If  $f = \text{false}$  or  $\neg f \in \text{Old}(q)$ 
  Then
    Return NodeList
  Else
    Create New Node  $q'$  s.t.
      Incoming( $q'$ ) := Incoming( $q$ )
      Old( $q'$ ) := Old( $q$ )  $\cup$  { $f$ }
      New( $q'$ ) := New( $q$ ) - { $f$ }
      Next( $q'$ ) := Next( $q$ )
    Return expand( $q'$ , NodeList)

```

Si  $f$  es la constante "false" o la negación de  $f$  ya fue procesada en  $q$ , no tenemos que hacer nada y retornamos la NodeList

En ese caso, reemplazamos el nodo  $q$  con un nodo  $q'$  donde "pasamos" la fórmula  $f$  de "New" a "Old", es decir, marcamos a la fórmula  $f$  como "procesada".

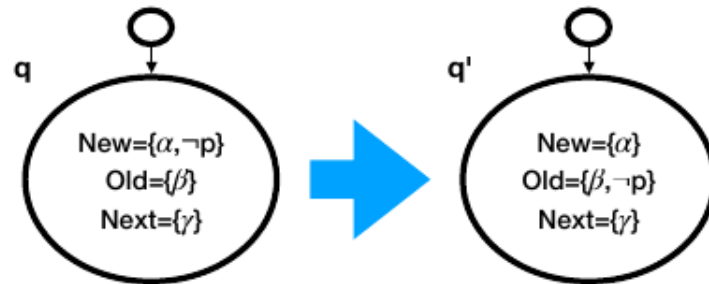


Figura 28: Caso  $f$  primitiva/literal.

ElseIf  $f \equiv h \vee k$

Then

```
Create two nodes q1, q2 such that
  Incoming(q1) := Incoming(q2) := Incoming(q)
  Old(q1) := Old(q2) := Old(q)  $\cup$  {hvk}
  New(q1) := (New(q) - {hvk})  $\cup$  {h}
  New(q2) := (New(q) - {hvk})  $\cup$  {k}
  Next(q1) := Next(q2) := Next(q)
Return expand(q2, expand(q1, NodeList))
```

Si  $f$  es una disjunción

Creo dos nodos  $q1, q2$  tales que comparten el mismo incoming y su Old es  $Old(q)$  y la fórmula  $\{h \vee k\}$ , donde  $q1$  procesará "h" y  $q2$  procesará "k"

Retorno expandir primero  $q1$ , y luego  $q2$ .

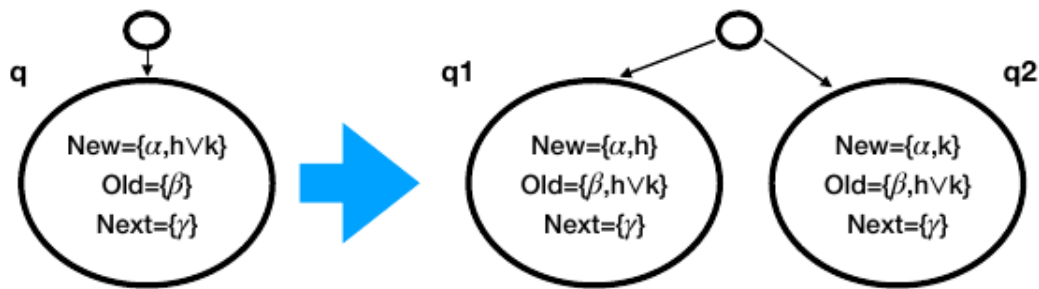
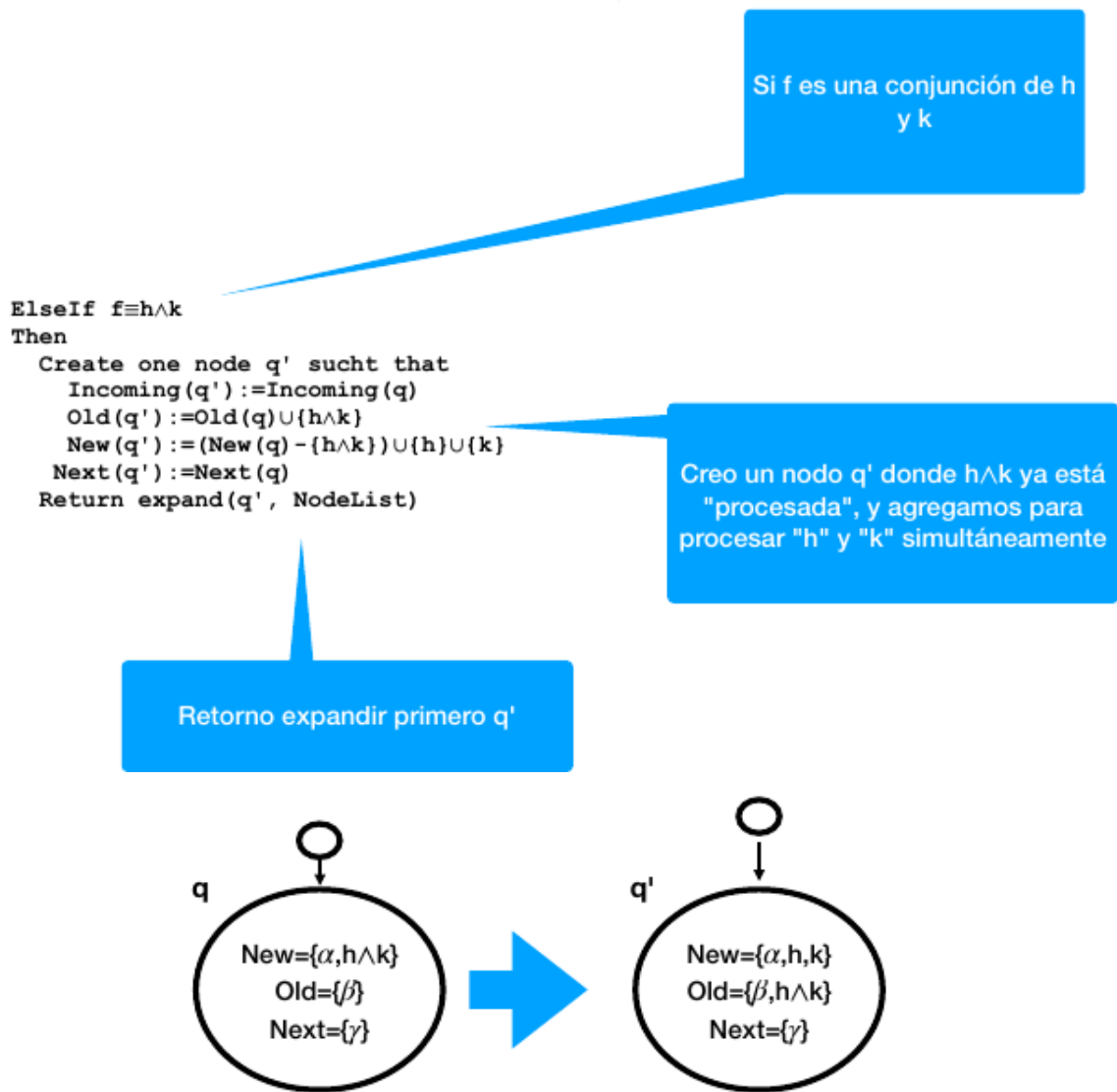


Figura 29: Caso  $f = h \vee k$ .

Figura 30: Caso  $f = h \wedge k$ .

ElseIf  $f \equiv X h$

Then

Create one node  $q'$  such that

$Incoming(q') := Incoming(q)$

$Old(q') := Old(q) \cup \{X h\}$

$New(q') := New(q) - \{X h\}$

$Next(q') := Next(q) \cup \{h\}$

Return  $expand(q', NodeList)$

Si  $f$  es "vale  $h$  en el siguiente estado"

Creo un nodo  $q'$  donde " $X h$ " ya está "procesada", y agregamos para procesar " $h$ " únicamente

Retorno expandir primero  $q'$

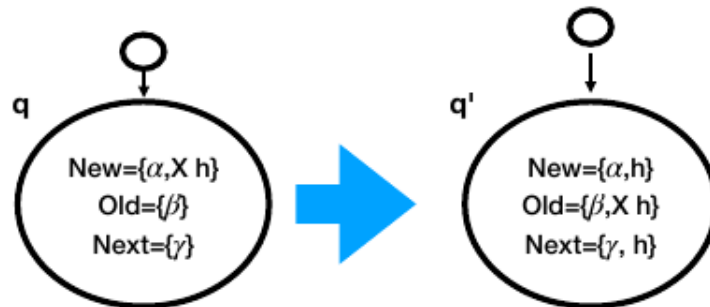
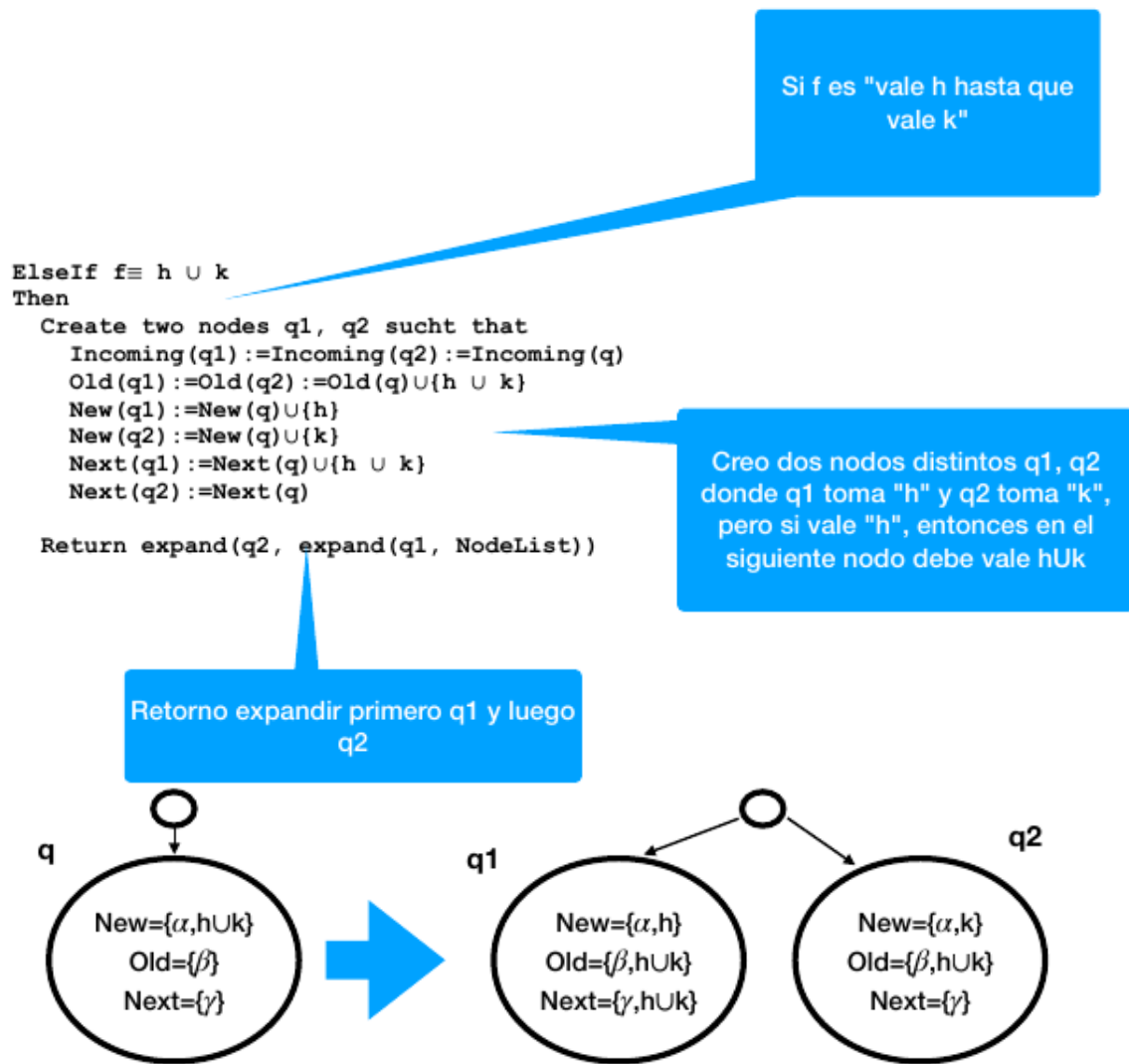


Figura 31: Caso  $f = X h$ .



Figura 32: Caso  $f = h \cup k$ .

```

ElseIf  $f \equiv h \text{ R } k$ 
Then

```

```

  Create two nodes  $q_1, q_2$  such that
  Incoming( $q_1$ ) := Incoming( $q_2$ ) := Incoming( $q$ )
  Old( $q_1$ ) := Old( $q$ )  $\cup$  { $h \text{ R } k$ }
  New( $q_1$ ) := New( $q$ )  $\cup$  { $h$ }  $\cup$  { $k$ }
  New( $q_2$ ) := New( $q$ )  $\cup$  { $k$ }
  Next( $q_1$ ) := Next( $q$ )
  Next( $q_2$ ) := Next( $q$ )  $\cup$  { $h \text{ R } k$ }

```

```

Return expand( $q_2$ , expand( $q_1$ , NodeList))

```

Si  $f$  es "vale  $h$  hasta que vale  $k$ "

Creo dos nodos distintos  $q_1, q_2$  donde  $q_1$  toma " $h$ " y " $k$ ", mientras que  $q_2$  toma solo " $k$ ", pero si vale " $k$ ", entonces en el siguiente nodo debe vale  $hRk$

Retorno expandir primero  $q_1$  y luego  $q_2$

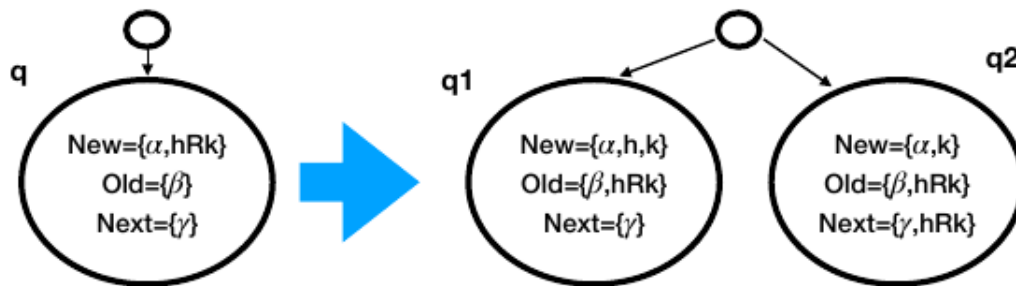


Figura 33: Caso  $f = h \text{ R } k$ .

Ejemplo: Aplicando el algoritmo para el caso  $P = a \cup b$ :

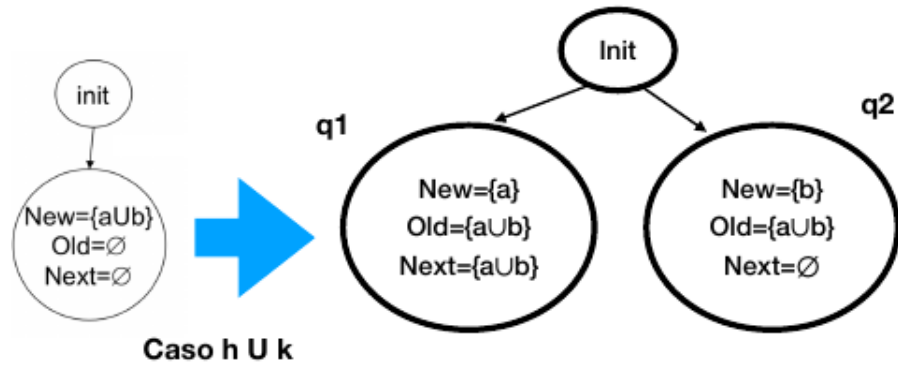


Figura 34: Ejemplo  $a \cup b$ , paso 1

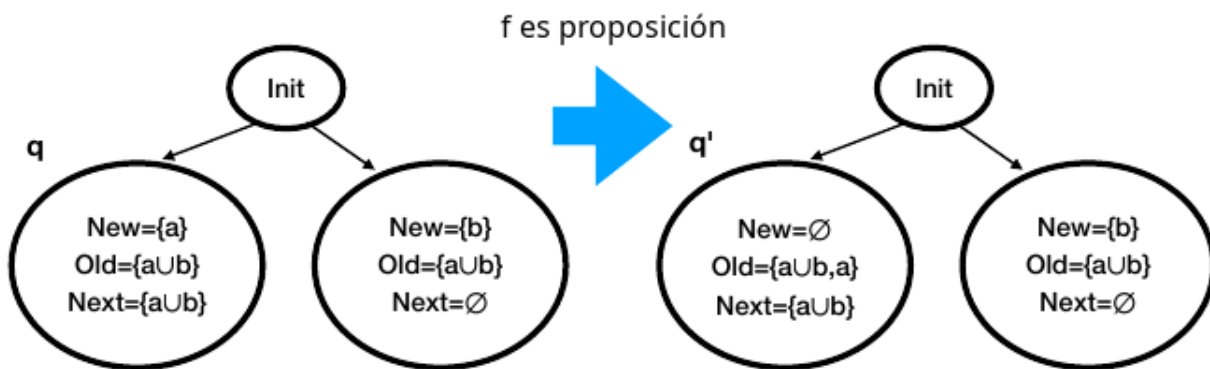
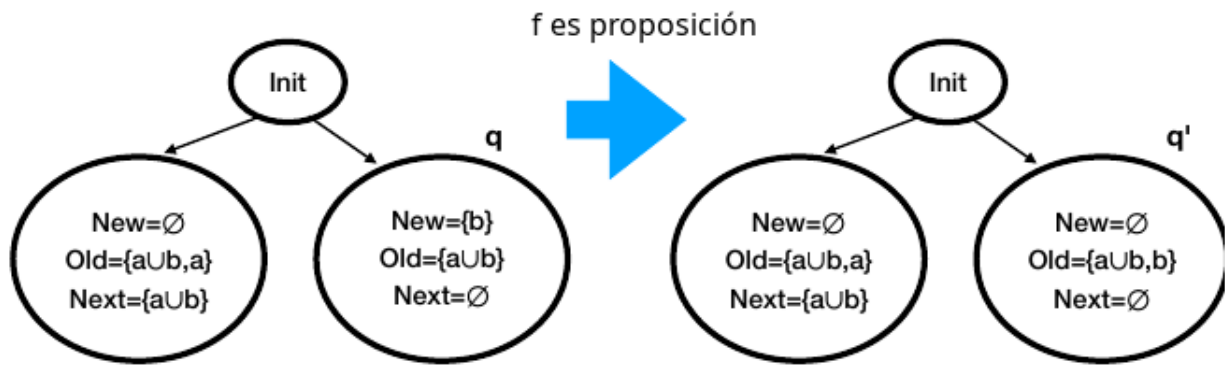
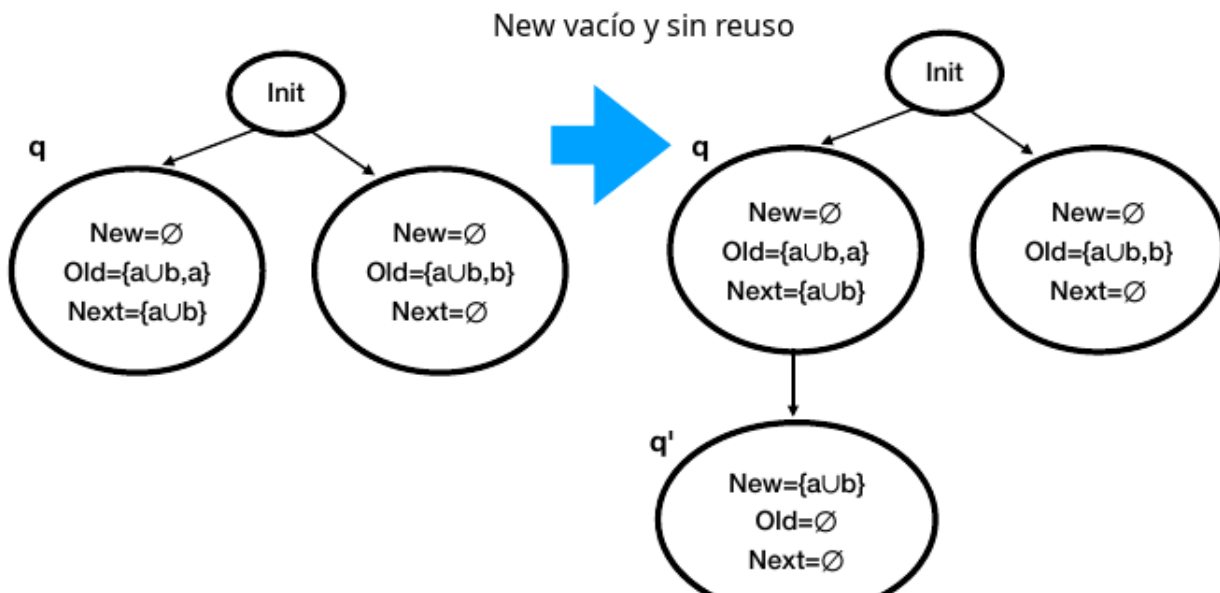
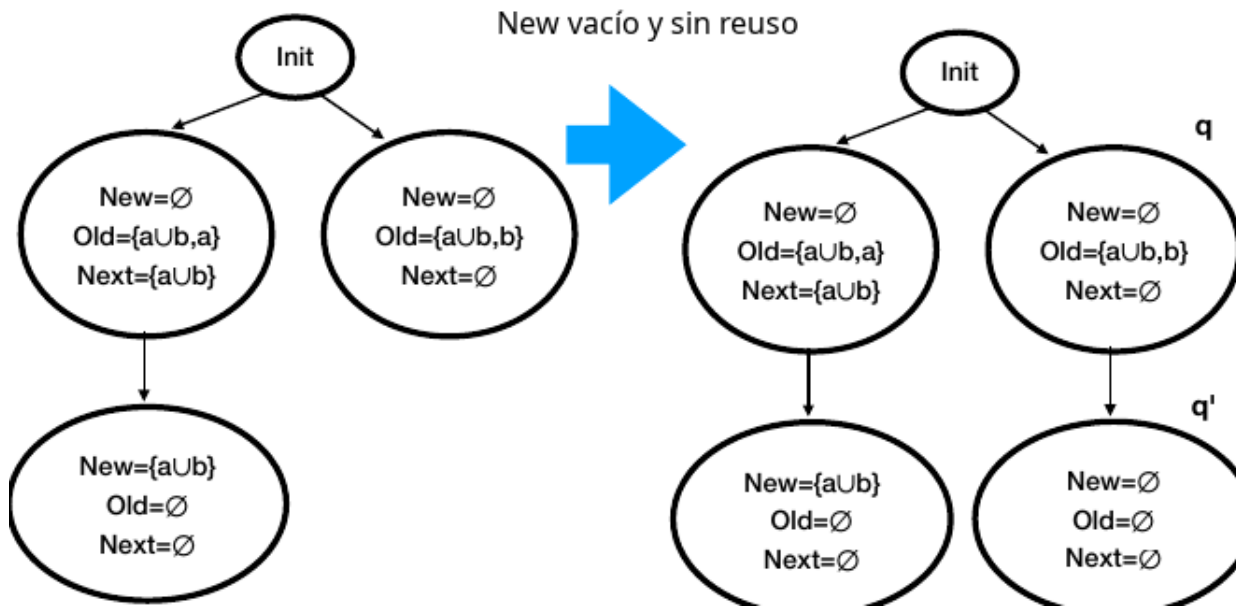
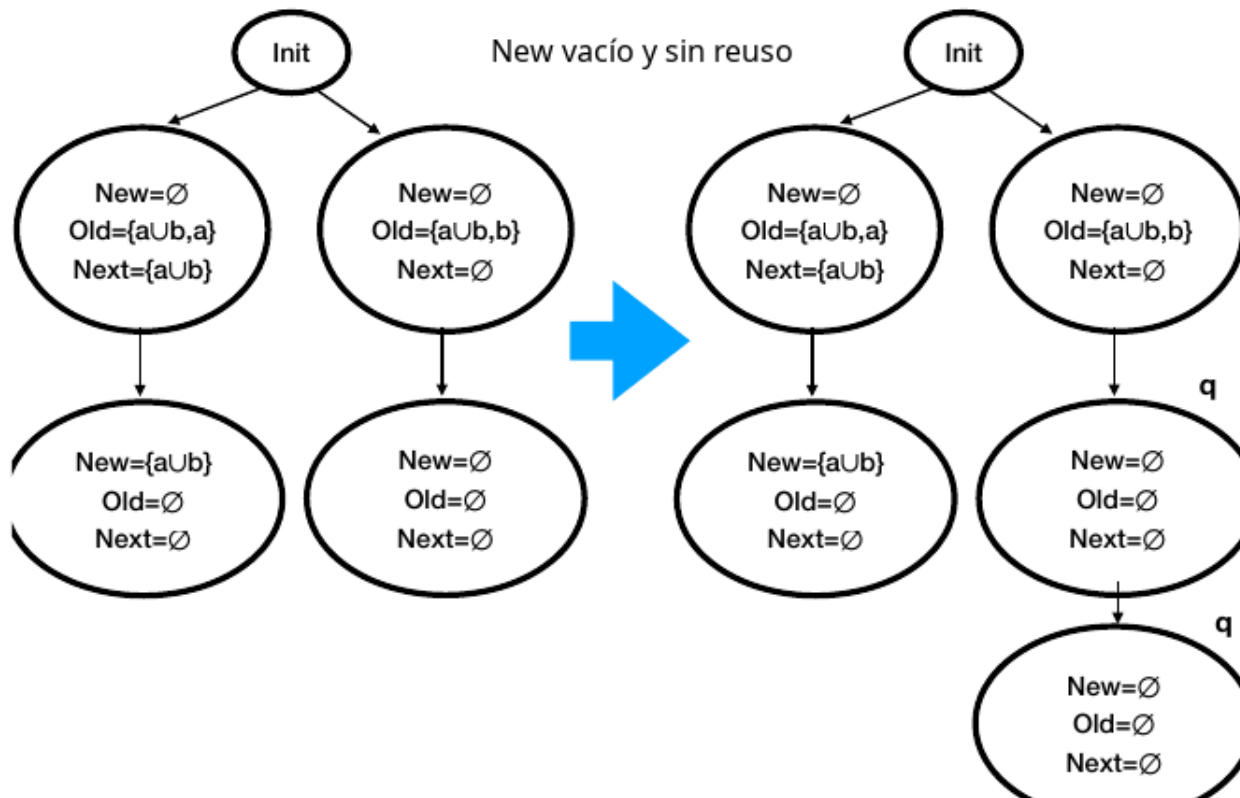
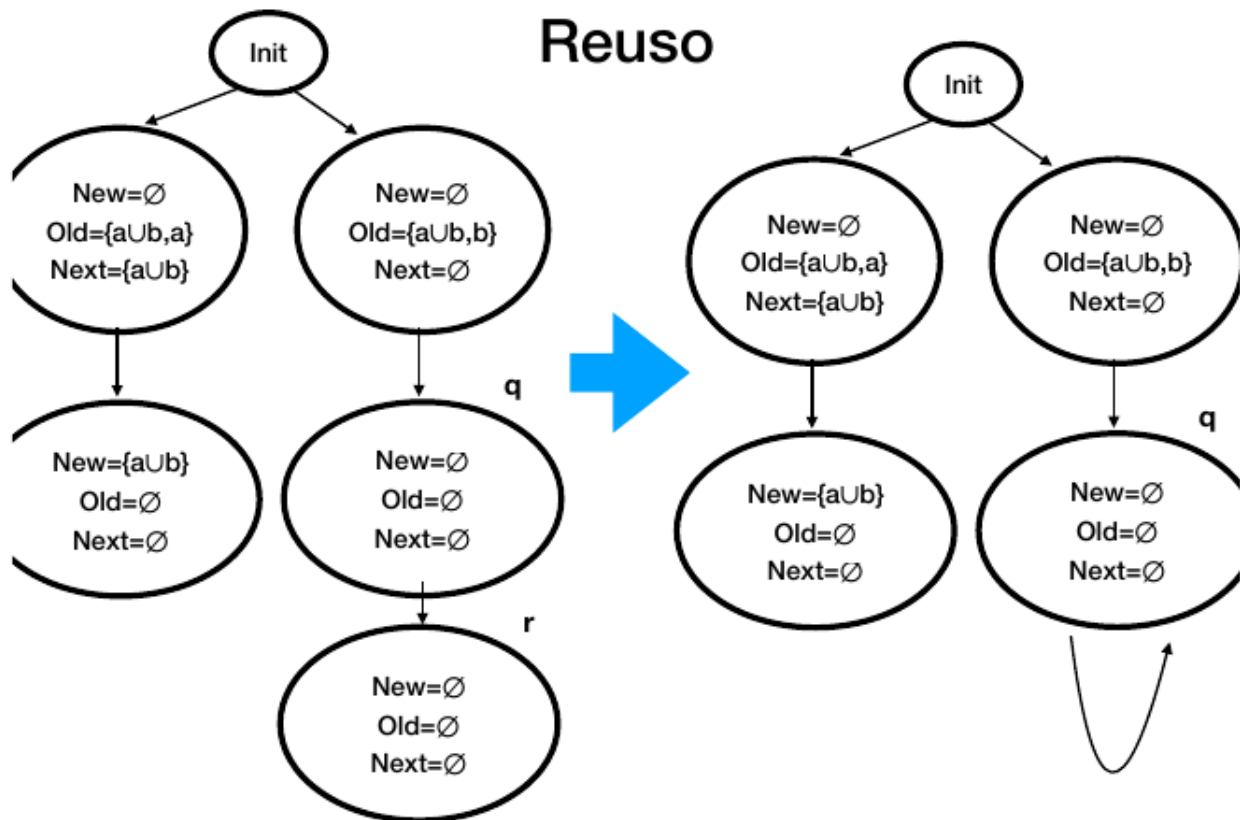


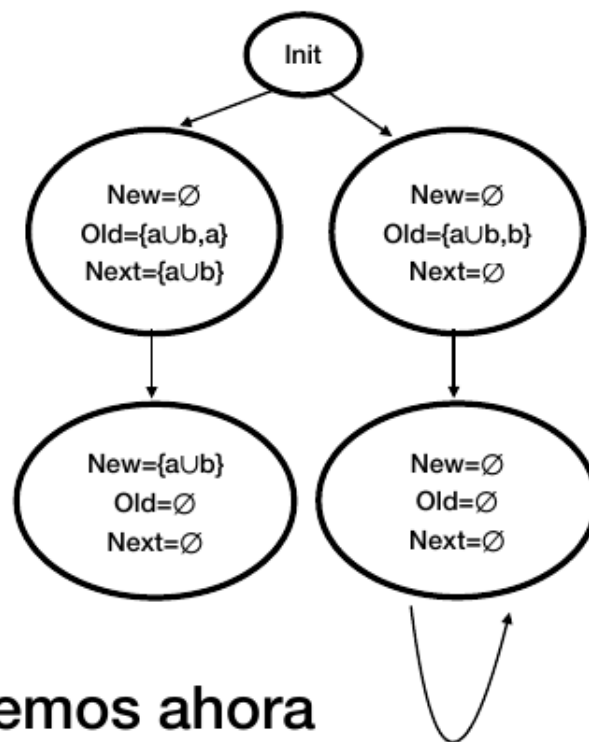
Figura 35: Ejemplo  $a \cup b$ , paso 2

Figura 36: Ejemplo  $a \cup b$ , paso 3Figura 37: Ejemplo  $a \cup b$ , paso 4

Figura 38: Ejemplo  $a \cup b$ , paso 5

Figura 39: Ejemplo  $a \cup b$ , paso 6

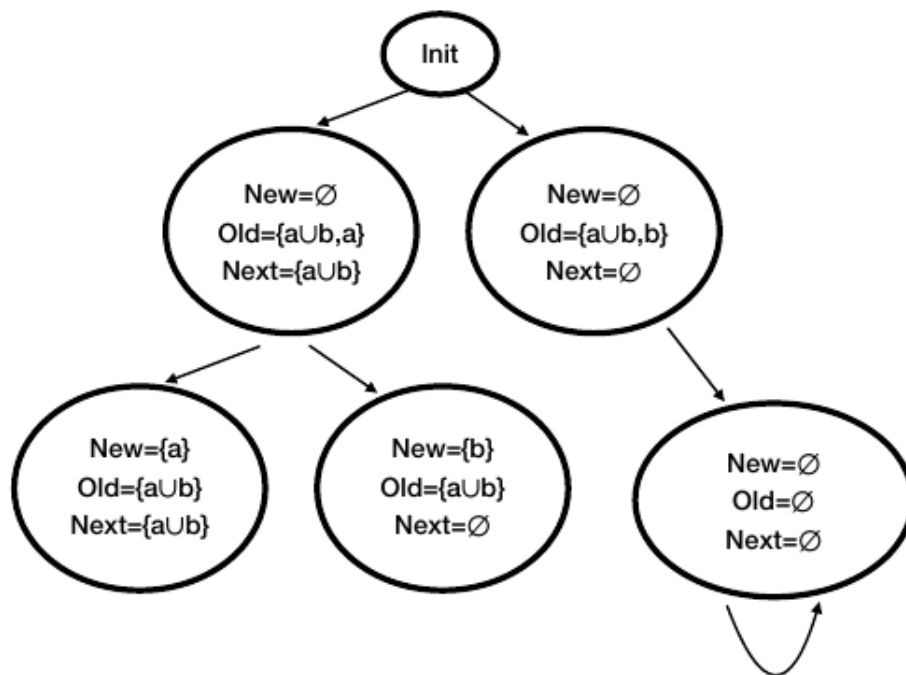
Figura 40: Ejemplo  $a \cup b$ , paso 7



**Tomemos ahora  
este autómeta**

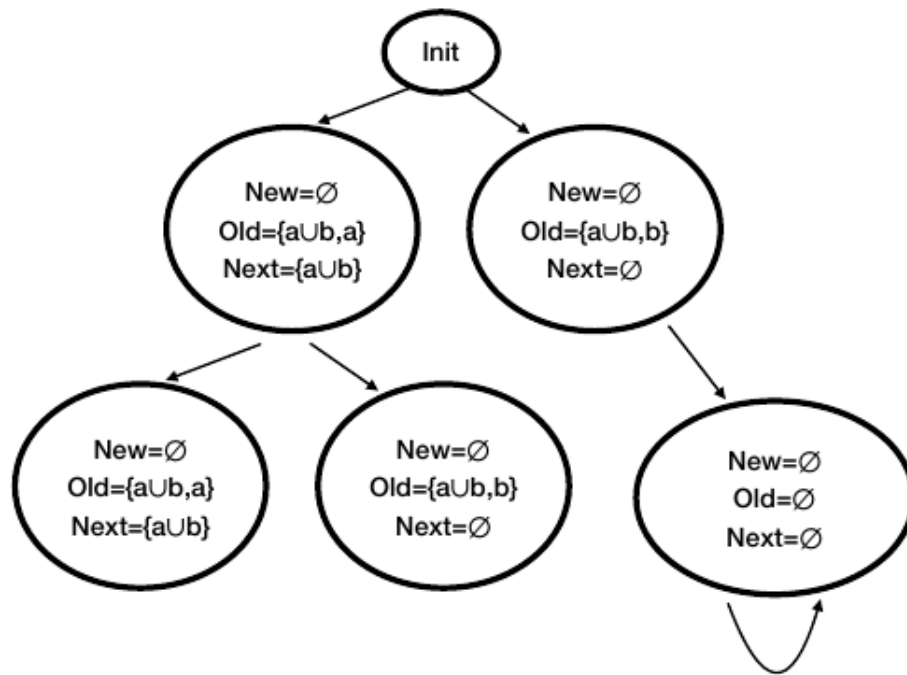
Figura 41: Ejemplo  $a \cup b$ , paso 8





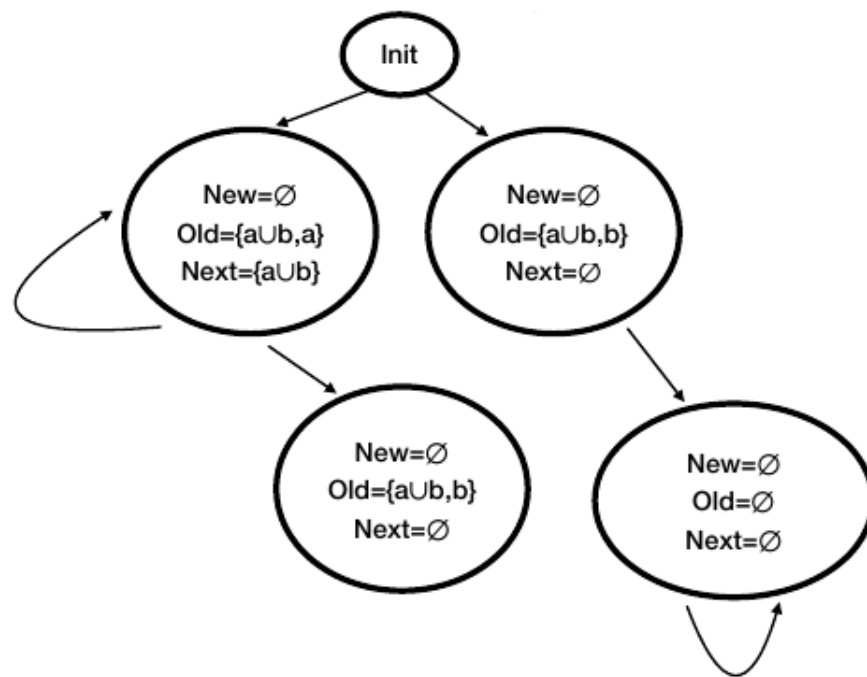
**Aplica nuevamente**  
 **$f = h \cup k$**

Figura 42: Ejemplo  $a \cup b$ , paso 9



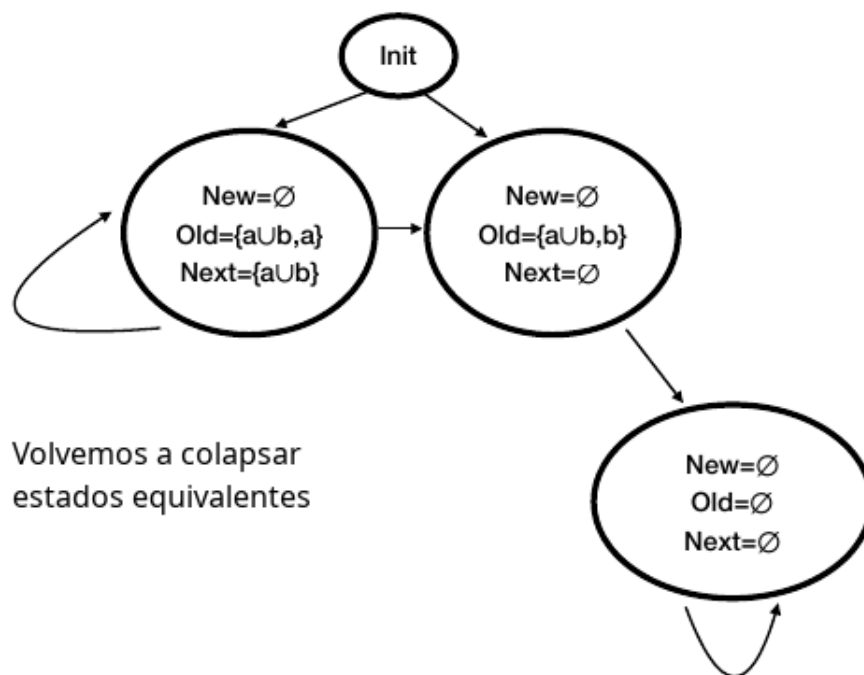
**Aplica  $f$  es proposición  
atómica**

Figura 43: Ejemplo  $a \cup b$ , paso 10



**Colapsamos estados  
equivalentes**

Figura 44: Ejemplo  $a \cup b$ , paso 11

Figura 45: Ejemplo  $a \cup b$ , paso 12

El paso 2 produce el siguiente autómata para el input "aUb"

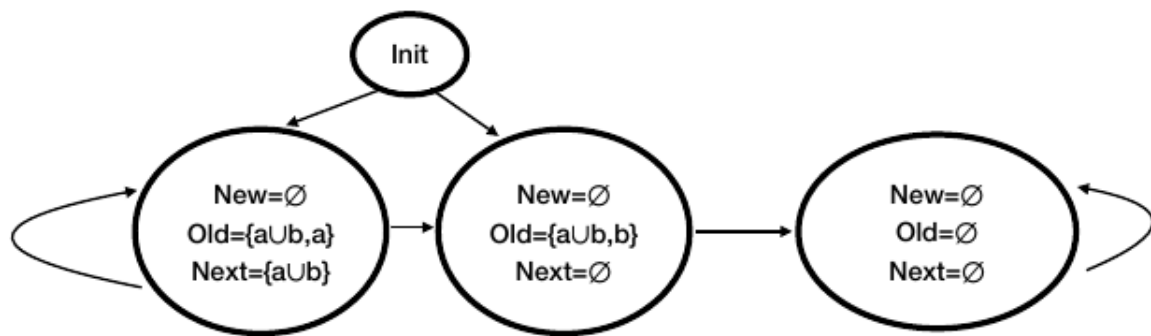


Figura 46: Ejemplo  $a \cup b$ , paso 13

- $(q,d,q') \in \Delta$  ssi  $q \in \text{incoming}(q')$  y  $d$  satisface la conjunción de las proposiciones negadas y no negadas que están en  $\text{Old}(q')$

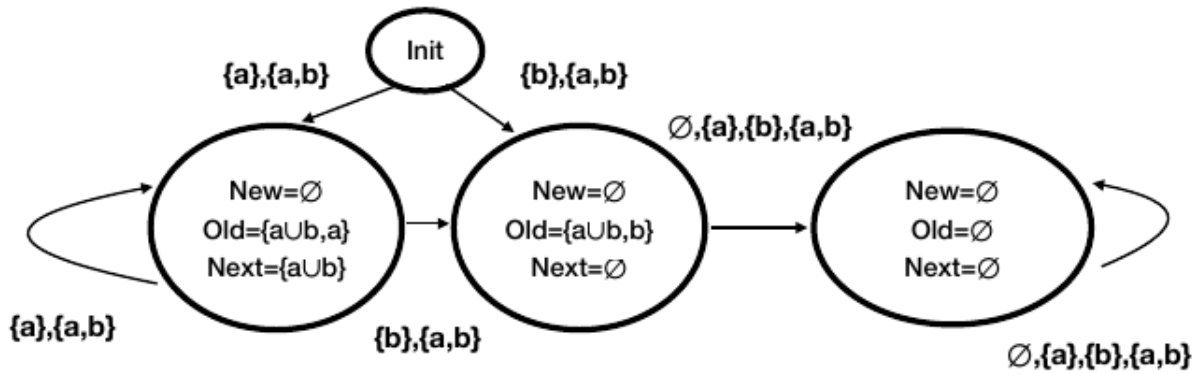
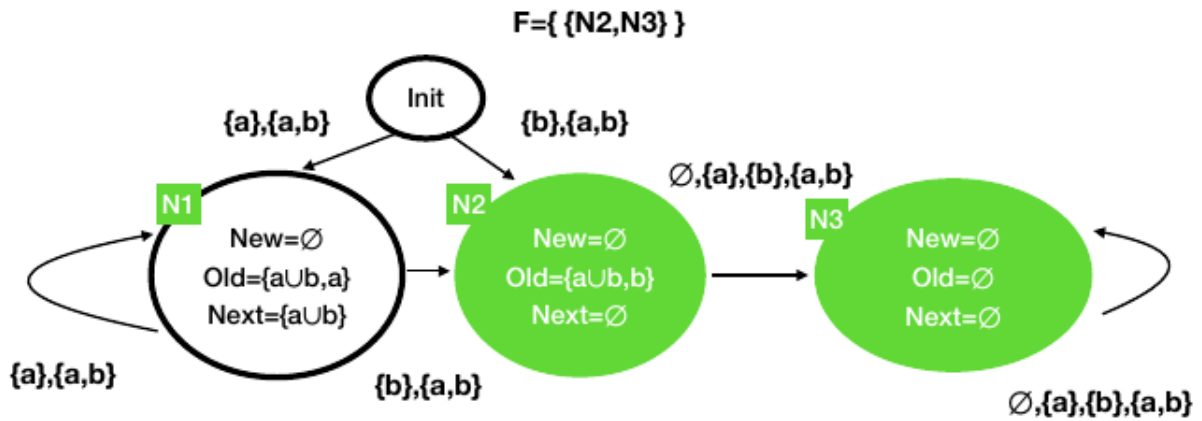
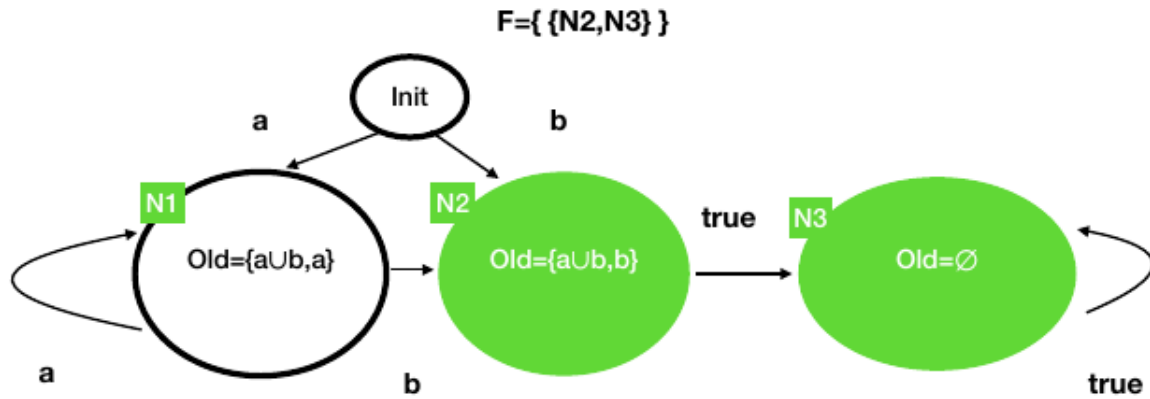


Figura 47: Ejemplo  $a \text{ U } b$ , paso 14

- Para cada sub-fórmula  $h \cup k$  existe un estado de aceptación  $F_i$  que contiene todos los estados  $q$  tales que o bien " $k$ "  $\in \text{Old}(q)$  o bien " $h \cup k$ "  $\notin \text{Old}(q)$

Figura 48: Ejemplo  $a \cup b$ , paso 15

- Finalmente, podemos cambiar los conjuntos de valuaciones en una transición por una fórmula proposicional que los caracteriza
- Este es el autómata de Büchi generalizado que acepta las trazas que satisfacen la fórmula LTL " $a \text{ U } b$ "

Figura 49: Ejemplo  $a \text{ U } b$ , paso 16

### 10.5.5. Conversión de LTS a autómata de Büchi

Todo lo visto hasta ahora fue para el primer paso del algoritmo de model checking (convertir la fórmula LTL a un autómata de Büchi). El segundo paso es convertir el LTS  $M$  a un autómata de Büchi  $A_M$  que caracterice todas las trazas que contiene  $M$ .

El LTS corresponde a la descripción del comportamiento del programa concurrente, y esto hay que transformarlo en un autómata de Büchi que represente las mismas trazas. Este autómata se construye de la siguiente manera:

#### Algoritmo LTS2Büchi

- Sea un LTS  $M = \langle S, A, \Delta, s_0 \rangle$ , se define el autómata de Büchi  $A_M = \langle \Sigma, Q, \Delta', Q_0, F \rangle$  de la siguiente manera:
  - $\Sigma := Act$  (conjunto de acciones observables)
  - $Q := S$  es el conjunto de estados.
  - $\Delta' := \Delta$  es la relación de transición.
  - $Q_0 := \{s_0\}$  es el único estado inicial.
  - $F := S$  es el conjunto de estados de aceptación (son todos).

*Observación.* Todos los estados del LTS son estados de aceptación en el autómata de Büchi. Esto es así porque la noción de aceptación viene del lado de la propiedad LTL, no del modelo del sistema concurrente. Por lo tanto, no hay que descartar ninguna traza que podría ocurrir en el LTS.



### 10.5.6. Intersección de autómatas de Büchi

El siguiente paso del algoritmo de model checking es verificar que las trazas del autómata de la negación de la fórmula  $P$  y el programa  $M$  sean disjuntas. Para esto, hay que:

1. Hacer la intersección de los autómatas de Büchi  $A_{\neg P}$  y  $A_M$ .
2. Verificar que el autómata resultante reconozca el lenguaje vacío.

Primero se verá cómo obtener la intersección (o producto) de dos autómatas de Büchi:

- Sean  $A_1 = \langle \Sigma, Q_1, \Delta_1, Q_{01}, F_1 \rangle$  y  $A_2 = \langle \Sigma, Q_2, \Delta_2, Q_{02}, F_2 \rangle$  dos autómatas de Büchi. Se define el autómata de Büchi generalizado  $A_1 \times A_2 = \langle \Sigma, Q, \Delta, Q_0, F \rangle$  de la siguiente manera:

- $Q := Q_1 \times Q_2$
- $Q_0 := Q_{01} \times Q_{02}$
- $F := \{F_1 \times Q_2, Q_1 \times F_2\}$
- $((q_1, q_2), a, (q'_1, q'_2)) \in \Delta \iff (q_1, a, q'_1) \in \Delta_1 \wedge (q_2, a, q'_2) \in \Delta_2$

**Lema 10.1.**  $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$

### 10.5.7. Chequeo de vacuidad de lenguaje

Por último, para completar el algoritmo de model checking falta comprobar si el lenguaje reconocido por el autómata de Büchi de la intersección de  $A_{\neg P}$  y  $A_M$  es vacío. De ser así, el programa modelado por  $M$  cumple la propiedad  $P$ ; en caso contrario, no.

Recordar que un autómata de Büchi acepta una traza cuando existe una ejecución que visita un estado de aceptación infinitas veces.

El algoritmo consiste en:

- Buscar un ciclo en el autómata que contenga un estado de aceptación, y sea alcanzable desde el estado inicial (búsqueda de componentes fuertemente conexas alcanzables, que contengan estado de aceptación).
- Si existe ese ciclo, entonces hay una traza que es aceptada por el autómata (dado que se puede quedar *ciclando* ahí infinitamente). Por lo tanto, el lenguaje reconocido por el autómata de Büchi es no vacío.
- En caso de que no exista tal ciclo, el lenguaje aceptado es vacío.

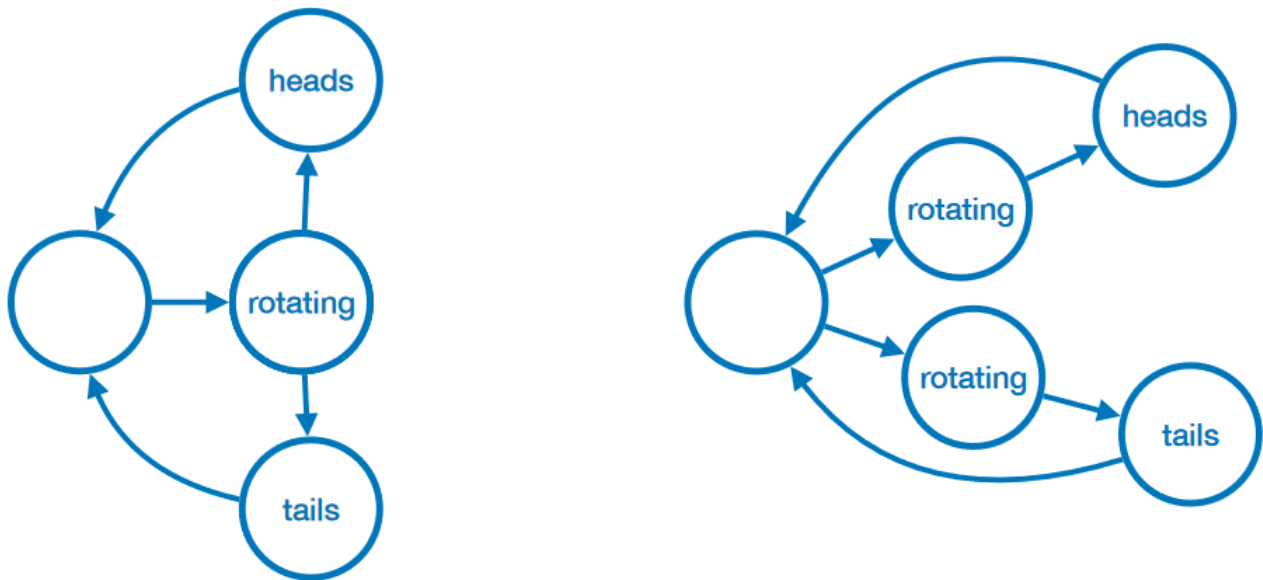
Existen dos algoritmos conocidos con complejidad lineal  $O(|Q| + |\Delta|)$  para calcular componentes fuertemente conexas en grafos: los algoritmos de Kosaraju y de Tarjan.

## 11. Model Checking usando CTL

### 11.1. Lógicas temporales de *branching*

Las lógicas temporales que consideran el tiempo de manera lineal (como LTL) *se quedan cortas* a la hora de describir ciertas propiedades que pueden presentar los sistemas reactivos:

**Ejemplo 11.1.** Considerar las siguientes estructuras de Kripke:



Es claro que tienen las mismas trazas, pero la de la izquierda es no determinística, mientras que la de la derecha no (parecido al ejemplo de la moneda en LTS). No es posible distinguir estas dos estructuras usando fórmulas LTL.

Entonces, una de las cosas que LTL no es capaz de capturar ante dos estructuras con las mismas trazas es el no determinismo.

De aquí surge la idea de considerar el paso del tiempo de una manera alternativa. En LTL se analizan las ejecuciones individualmente; lo que proponen las lógicas de branching (*ramificación*) es analizar el **cómputo como un árbol**, obteniendo un árbol de cómputo desenrollando los ciclos de un programa (notar que esto hará que los árboles tengan profundidad infinita).

Así, en vez de mirar todas las trazas por separado linealmente, se puede pensar que un sistema de transición representa un árbol de cómputo, mirando el tiempo de manera *ramificada*. En cda estado, se abre una rama por cada posible evolución del sistema desde dicho estado.

Observación. La noción de bisimulación también se puede definir, de manera alternativa, como un isomorfismo entre los árboles de cómputo de dos sistemas de transiciones.

### 11.2. Computational Tree Logic

Esta es una lógica de branching, donde el tiempo *se ramifica* hacia el futuro, representado por las ramas del árbol.

Se cuantifica sobre **camino en el árbol** de cómputo, con lo cual se agregan nuevos operadores:

- Operador universal de caminos:  $A$  (i.e. “para todo camino...”)
- Operador existencial de caminos:  $E$  (i.e. “existe un camino...”)

Además, se usan todos los operadores que se usaban en LTL, para cuantificar sobre estados:

- Siguiente estado:  $X$
- Necesidad:  $G$  (lo mismo que  $\Box$  en LTL)
- Eventualidad:  $F$  (lo mismo que  $\Diamond$  en LTL)
- Until:  $U$

Con lo cual, los operadores temporales son combinados. Por ejemplo,  $AG$ ,  $AF$ ,  $AU$ ,  $EG$ ,  $EX$ , etc.

**Ejemplo 11.2.**  $AG p$ : “en todos los caminos (desde el estado inicial), vale que en todos sus estados vale  $p$ ”.

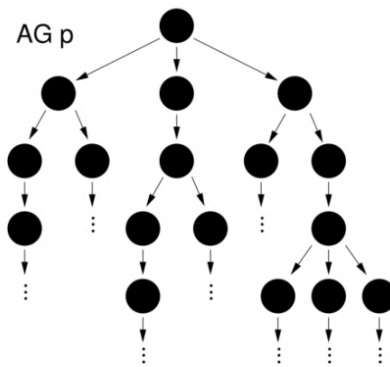


Figura 50: En los estados negros vale  $p$

**Ejemplo 11.3.**  $AF p$ : “en todos los caminos, vale existe un estado donde vale  $p$ ”.

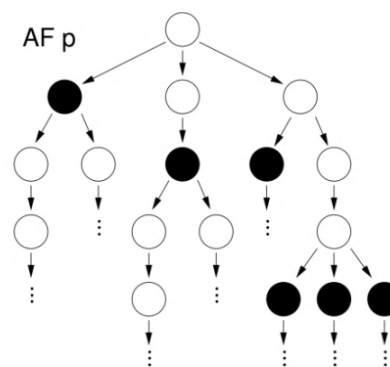


Figura 51: En los estados negros vale  $p$

**Ejemplo 11.4.**  $AX\ p$ : “en todos los caminos, vale que en el siguiente estado vale  $p$ ”.

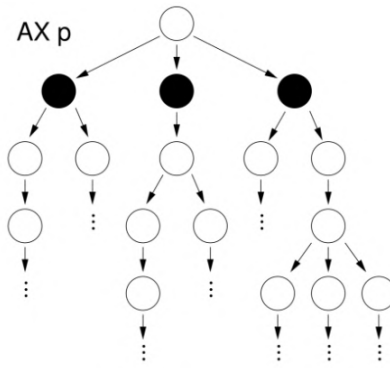


Figura 52: En los estados negros vale  $p$

**Ejemplo 11.5.**  $p\ AU\ q$ : “en todos los caminos, vale que vale  $p$  hasta que vale  $q$ ”.

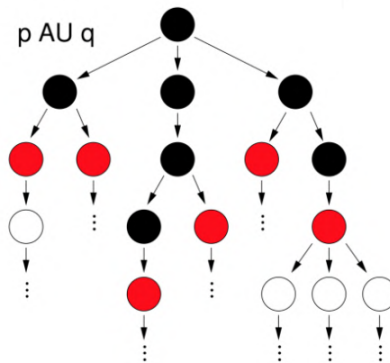


Figura 53: En los estados negros vale  $p$ , y en los rojos vale  $q$ .

**Ejemplo 11.6.**  $EG\ p$ : “existe un camino tal que en todos sus estados vale  $p$ ”.

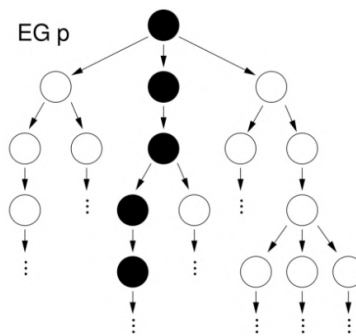


Figura 54: En los estados negros vale  $p$

**Ejemplo 11.7.**  $EF p$ : “existe un camino tal que en algún estado vale  $p$ ”.

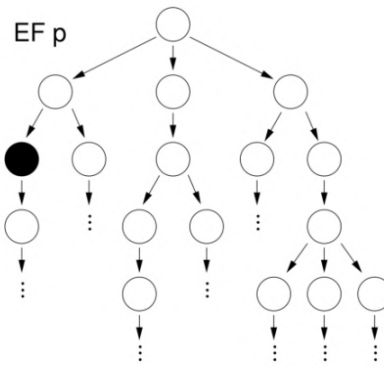


Figura 55: En los estados negros vale  $p$

**Ejemplo 11.8.**  $EX p$ : “existe un camino tal que en el próximo estado vale  $p$ ”.

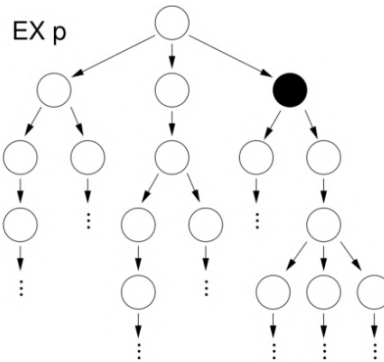


Figura 56: En los estados negros vale  $p$

**Ejemplo 11.9.**  $p EU q$ : “existe un camino tal que vale  $p$  hasta que vale  $q$ ”.

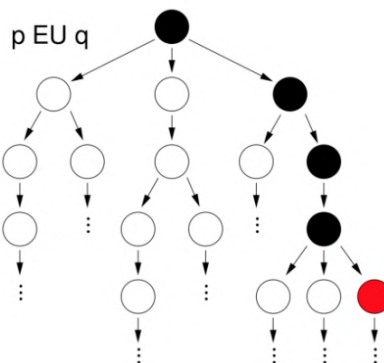


Figura 57: En los estados negros vale  $p$ , y en los rojos vale  $q$ .

### 11.3. Semántica de CTL

Se dice que un árbol de cómputo infinito satisface una fórmula si su raíz la satisface.

A continuación se muestra la interpretación de cada operador (es la misma idea que en LTL, pero con un nivel adicional de cuantificadores).

Sea  $p$  una proposición, y  $\alpha, \beta$  fórmulas CTL; sea  $v$  una valuación que asigna *verdadero* ( $T$ ) o *falso* ( $F$ ) a cada proposición.

- $s \models p$  sii  $v(s, p) = T$
- $s \models \neg\alpha$  sii  $s \not\models \alpha$
- $s \models \alpha \wedge \beta$  sii  $s \models \alpha$  y  $s \models \beta$
- $s \models \alpha \vee \beta$  sii  $s \models \alpha$  o  $s \models \beta$
- $s_0 \models EX \alpha$  sii existe un camino  $s_0, s_1, \dots$  tal que  $s_1 \models \alpha$
- $s_0 \models AX \alpha$  sii para todo camino  $s_0, s_1, \dots$ ,  $s_1 \models \alpha$
- $s_0 \models EG \alpha$  sii existe un camino  $s_0, s_1, \dots$  tal que  $\forall i \geq 0 : s_i \models \alpha$
- $s_0 \models AG \alpha$  sii para todo camino  $s_0, s_1, \dots$ ,  $\forall i \geq 0 : s_i \models \alpha$
- $s_0 \models EF \alpha$  sii existe un camino  $s_0, s_1, \dots$  tal que  $\exists i \geq 0 : s_i \models \alpha$
- $s_0 \models AF \alpha$  sii para todo camino  $s_0, s_1, \dots$ ,  $\exists i \geq 0 : s_i \models \alpha$
- $s_0 \models \alpha EU \beta$  sii existe un camino  $s_0, s_1, \dots$  tal que  $\exists i \geq 0 : s_i \models \beta$  y  $\forall 0 \leq j < i : s_j \models \alpha$
- $s_0 \models \alpha AU \beta$  sii para todo camino  $s_0, s_1, \dots$ ,  $\exists i \geq 0 : s_i \models \beta$  y  $\forall 0 \leq j < i : s_j \models \alpha$

## 11.4. Expresividad de CTL y LTL

Recordar que en el Ejemplo 11.1 se vio que no era posible distinguir los sistemas de transiciones usando fórmulas LTL, ya que estas eran incapaces de expresar la diferencia entre el determinismo y el no determinismo.

Sin embargo, esto sí es posible con fórmulas CTL: el sistema de la izquierda cumple la siguiente propiedad, mientras que el de la derecha no:

$$AG(\text{rotating} \rightarrow (EX \text{heads} \wedge EX \text{tails}))$$

La clave aquí es que con CTL se puede describir la **potencialidad** que tienen los estados en donde acaba de ocurrir **rotating**.

### 11.4.1. CTL > LTL

El poder expresivo de CTL es mayor que el de LTL en el sentido de que CTL puede **razonar sobre múltiples trazas simultáneamente**, y **sobre comportamiento potencial** (mirar cada rama que se abre desde un estado dado, y ver si alguna/todas se comportan de cierta manera), mientras LTL se ve limitada al considerar cada traza individualmente y de manera lineal, no ramificada.

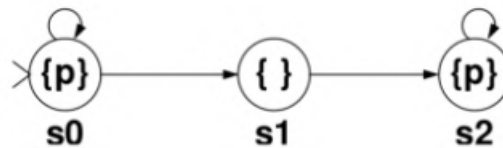
Algunos posibles escenarios de uso para CTL, que no son factibles en LTL, son:

- $AGEF\alpha$ : dice que siempre va a existir la *posibilidad* de que valga  $\alpha$ . Esto es distinto de la propiedad “siempre va a valer  $\alpha$ ”. Por ejemplo, si  $\alpha = \text{abort}$ , esto podría estar expresando que siempre va a ser posible abortar el proceso.
- $AG(EX\alpha \Leftrightarrow EX\beta)$ : en todo momento, si el sistema ofrece la opción  $\alpha$  (e.g. un botón “Aceptar”), entonces también ofrecerá la posibilidad  $\beta$  (e.g. un botón “Cancelar”). Esto es una manera de asegurar que el usuario siempre pueda aceptar o cancelar una acción.
- $AG(\rho \rightarrow EX(\alpha_1 \wedge EX(\alpha_2 \dots)))$ : esto se llama *caso de uso*, y dice que cuando ocurra una cierta precondition  $\rho$ , da lugar a una serie de pasos  $\alpha_1, \alpha_2, \dots$  que se debe poder efectuar (pero no necesariamente se deben efectuar de esa manera). De esta manera, se puede especificar distintas funcionalidades de un sistema, admitiendo la posibilidad de que un usuario use el sistema de diferentes formas. Por ejemplo, la precondition podría ser que “el usuario está autenticado”, y la secuencia de pasos algo como “hacer un *log in*”, luego “acceder al resumen de la tarjeta de crédito”, etc.

#### 11.4.2. LTL > CTL

Por otro lado, hay propiedades que se pueden expresar en LTL pero no en CTL:

**Ejemplo 11.10.** Considerar el siguiente sistema de transiciones:



Este sistema satisface la fórmula LTL  $\Diamond\Box p$ . Pero expresar esto mismo es imposible en CTL; el mejor intento podría ser algo de la forma  $AG\,AG\,p$ , pero esto no se cumple. En el árbol de cómputo, hay un camino  $(s_0)^\omega$  donde siempre está la posibilidad de ir a  $s_1$ , donde no vale  $p$ . Por lo tanto, no es verdad que en ese camino existe un estado en donde a partir de allí siempre vale  $p$  (que es lo que dice la fórmula CTL). Luego, la propiedad CTL no vale.

La conclusión es que LTL y CTL son **incomparables** en cuanto a expresividad.

### 11.5. Algoritmo de verificación para programas concurrentes

La idea será ahora definir un algoritmo que verifique automáticamente si un sistema concurrente satisface una propiedad CTL. El sistema estará representado por una estructura de Kripke, y lo que hay que verificar es si su árbol de cómputo cumple una fórmula CTL desde el estado inicial.

Hay dos enfoques para llevar esto a acabo:

- Representación explícita de estados de la estructura de Kripke (similar a lo que se hizo para LTL).
- Representación simbólica de estados

En la materia, se ve el primer enfoque: **model checking explícito de CTL**. Para eso, se redefine la semántica de CTL de una manera conveniente: los conjuntos característicos. Dada una fórmula, se define su semántica como el conjunto de estados de la estructura de Kripke que la satisfacen.

### 11.5.1. Una base de operadores CTL

Antes de esto, notar que no es necesario definir semánticamente todos los operadores CTL, dado que se puede reescribir varios de ellos en función de otros. En concreto, basta con usar  $\{\neg, \wedge, EX, EG, EU\}$ , puesto que:

$$\begin{aligned} EF \alpha &\equiv True EU \alpha \\ AX \alpha &\equiv \neg EX \neg \alpha \\ AG \alpha &\equiv \neg EF \neg \alpha \\ AF \alpha &\equiv \neg EG \neg \alpha \\ \alpha AW \beta &\equiv \neg (\neg \beta EU \neg (\alpha \vee \beta)) \\ \alpha AU \beta &\equiv AF \beta \wedge (\alpha AW \beta) \\ \alpha EW \beta &\equiv EG \alpha \vee (\alpha EU \beta) \end{aligned}$$

### 11.5.2. Semántica CTL con respecto a una estructura de Kripke

Sea  $\mathcal{K} = \langle S, s_0, \rightarrow, v \rangle$  una estructura de Kripke. Se define la semántica de cada fórmula CTL  $\alpha$  con respecto a  $\mathcal{K}$  como un conjunto de estados  $\llbracket \alpha \rrbracket_{\mathcal{K}}$  de la siguiente manera:

$$\begin{aligned} \llbracket p \rrbracket_{\mathcal{K}} &= v(p) \text{ para } p \in AP \\ \llbracket \neg \alpha \rrbracket_{\mathcal{K}} &= S \setminus \llbracket \alpha \rrbracket_{\mathcal{K}} \\ \llbracket \alpha \vee \beta \rrbracket_{\mathcal{K}} &= \llbracket \alpha \rrbracket_{\mathcal{K}} \cup \llbracket \beta \rrbracket_{\mathcal{K}} \\ \llbracket EX \alpha \rrbracket_{\mathcal{K}} &= \{s \mid \exists t : s \rightarrow t \wedge t \in \llbracket \alpha \rrbracket_{\mathcal{K}}\} \\ \llbracket EG \alpha \rrbracket_{\mathcal{K}} &= \{s \mid \exists \rho \text{ traza} : \rho[0] = s \wedge \forall i \geq 0, \rho[i] \in \llbracket \alpha \rrbracket_{\mathcal{K}}\} \\ \llbracket \alpha EU \beta \rrbracket_{\mathcal{K}} &= \{s \mid \exists \rho \text{ traza} : \rho[0] = s \wedge \exists k \geq 0 : \forall i < k, \rho[i] \in \llbracket \alpha \rrbracket_{\mathcal{K}} \wedge \rho[k] \in \llbracket \beta \rrbracket_{\mathcal{K}}\} \end{aligned}$$

Se dice que  $\mathcal{K}$  satisface  $\alpha$  (notado  $\mathcal{K} \models \alpha$ ) si y sólo si  $s_0 \in \llbracket \alpha \rrbracket_{\mathcal{K}}$ .

### 11.5.3. El algoritmo

Dada una estructura de Kripke  $\langle S, s_0, R, v \rangle$  y una fórmula CTL  $P$ :

1. Traducir  $P$  a la base  $\{\neg, \wedge, EX, EG, EU\}$ .
2. Construir los conjuntos característicos de manera *bottom-up*.

Cada operador se analiza una vez, y es resuelto en  $O(|S| + |R|)$ . Esto es una vez por operador, así que la complejidad temporal del algoritmo es de  $O(|P| \times (|S| + |R|))$ . Esto es mejor que el  $O(2^{|P|} \times (|S| + |R|))$  de LTL; pero recordar que  $|S|$  explota exponencialmente: si la estructura de Kripke viene de la composición paralela de varios sistemas de transición, entonces la cantidad de estados será exponencial respecto a la cantidad de sistemas compuestos.



Para la construcción *bottom-up* de conjuntos característicos, se construyen los conjuntos para cada sub-fórmula, y a partir de ellos se van construyendo los de las fórmulas que contienen a dichas sub-fórmulas.

Los operadores se analizan desde los niveles inferiores en el árbol sintáctico de la fórmula, y se va subiendo a medida que son resueltos.

Ahora se verá la construcción para cada operador:

- $\llbracket p \rrbracket_{\mathcal{K}}$ : para proposiciones y operadores booleanos es fácil: basta con mirar todos los estados e incluir aquellos que tienen  $v(p) = T$

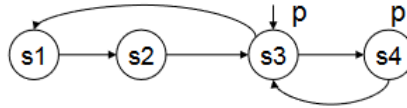


Figura 58: Ejemplo para  $\llbracket p \rrbracket_{\mathcal{K}}$

- $\llbracket EX p \rrbracket_{\mathcal{K}}$ : tomar todos los estados de  $\llbracket p \rrbracket_{\mathcal{K}}$ , y dar un *paso hacia atrás* en cada estado (incluyendo entonces los estados que están *una transición atrás* de los de  $p$ ).

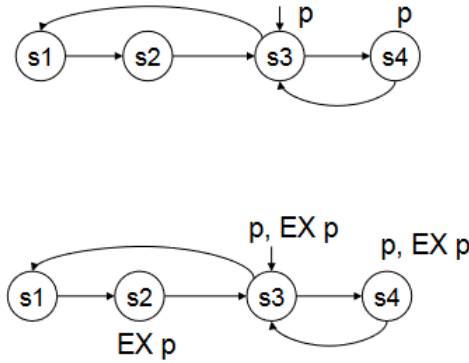
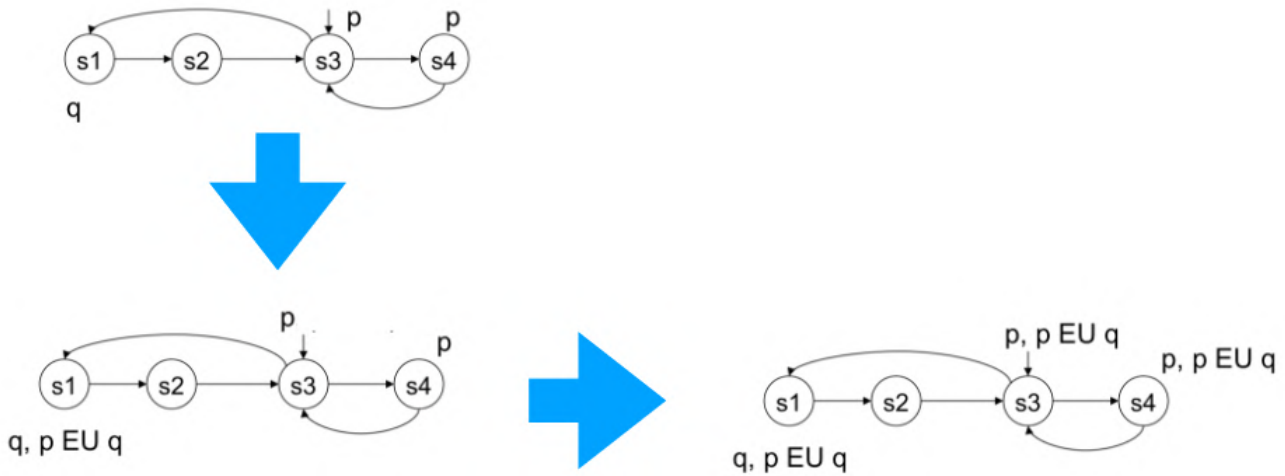
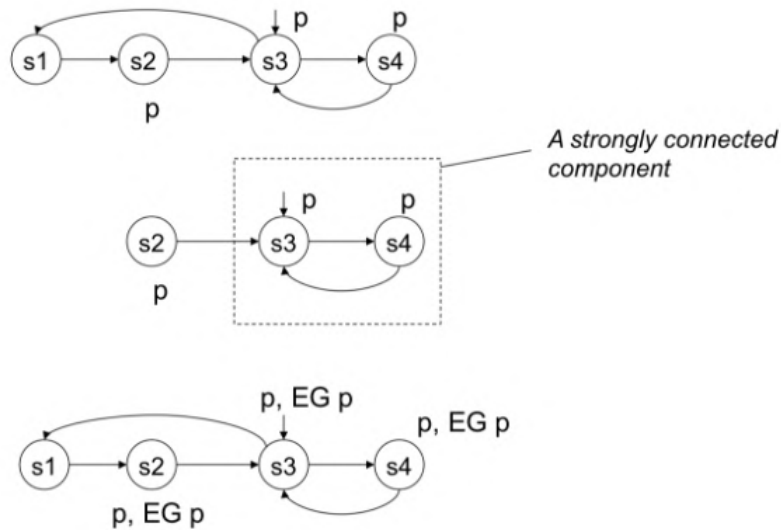


Figura 59: Ejemplo para  $\llbracket EX p \rrbracket_{\mathcal{K}}$

- $\llbracket p \text{ EU } q \rrbracket_{\mathcal{K}}$ : etiquetar como  $p \text{ EU } q$  a todos los estados donde vale  $q$ , y luego dar *pasos hacia atrás* mientras valga  $p$ , e incluir sólo esos en el conjunto.

Figura 60: Ejemplo para  $\llbracket p \text{ EU } q \rrbracket_{\mathcal{K}}$ 

- $\llbracket EG p \rrbracket_{\mathcal{K}}$ : primero eliminar los estados donde no vale  $p$  (y las transiciones que entran y o salen de ellos). Luego, calcular las componentes fuertemente conexas y marcar los estados donde vale  $p$  y desde los cuales se puede alcanzar las componentes calculadas (incluidos aquellos estados dentro de ellas).

Figura 61: Ejemplo para  $\llbracket EG p \rrbracket_{\mathcal{K}}$ 

#### 11.5.4. Limitaciones del model checking explícito

Se necesita una representación explícita para el espacio de estados del problema, dado que hay que utilizar algoritmos sobre grafos para el cálculo de los conjuntos característicos.

Si el sistema es composición de procesos concurrentes, hay un problema de escalabilidad.